FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Representação de aplicações C/C++ segundo um grafo para execução em sistemas heterogéneos

João Manuel Ferreira Trindade

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Jorge Gomes Barbosa, Prof. Dr.

Junho de 2016

Representação de aplicações C/C++ segundo um grafo para execução em sistemas heterogéneos

João Manuel Ferreira Trindade

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Rui Carlos Camacho de Sousa Ferreira da Silva (Prof. Dr.)

Vogal Externo: João Luís Ferreira Sobral (Prof. Dr.)

Orientador: Jorge Manuel Gomes Barbosa (Prof. Dr)

_____

30 de Junho de 2016

Resumo

São cada vez mais as áreas de investigação que recorrem a sistemas computacionais de elevada performance para resolver complexos modelos matemáticos.

Para dar resposta a problemas cada vez mais complexos, tem havido um grande desenvolvimento no *hardware* para os sistemas computacionais de elevado desempenho, tecnologias como CUDA - *Compute Unified Device Architecture* - ou FPGA - *Field-programmable gate array* - têm-se tornado parte integrante dos tradicionais *clusters* informáticos, dando assim origem a sistemas denominados heterogéneos. Dadas as especificidades deste tipo de infraestrutura, é necessário repensar a forma de execução de tarefas, de forma a garantir um proveito significativo das novas soluções: conseguir mapear inteligentemente tarefas com os recursos que melhor as consigam resolver permitirá obter, não só um melhor desempenho computacional, mas também reduzir custos energéticos do sistema.

Esta dissertação pretende explorar uma forma de representação de código C/C++ através de grafos dirigidos que permitam posteriormente a utilização de algoritmos de escalonamento para sistemas heterogéneos (como PEFT, HEFT, entre outros). Para isso, propõe-se uma análise e identificação de regiões criticas através da árvore sintática abstrata (AST) obtida da compilação do código com o compilador Clang. Esta análise, juntamente com informação obtida de modelos computacionais, permitirá então gerar perfis otimizados para a execução do código.

A ferramenta obtida, e descrita nesta dissertação, oferece uma forma de obtenção de grafos dirigidos a partir de Árvores Sintáticas Abstratas do compilador, e identifica regiões onde é possível a utilização de paralelismo. Esta análise à especificidade do código fonte possibilita uma posterior utilização de algoritmos de escalonamento que otimizam a sua execução para uma arquitetura heterogénea.

Abstract

It has become more and more commonplace for research areas to resort to High Performance Computing systems to resolve complex mathematical models.

In order to resolve the ever-increasing complexity of these models, there have been significant advances in hardware for high performance computing systems – namely technologies such as CUDA – Compute Unified Device Architecture – or FPGA – Field-programmable gate arrays. These have become an integral part of the traditional computing clusters, giving rise to heterogeneous systems. Given the specificity of these infrastructures, it is necessary to rethink the manner in which the tasks are executed so as to ensure significant advantages are gained from these new solutions: enabling intelligent task mapping with the resources best able to resolve them will not only ensure better computing performance, but will also reduce system energy requirements.

This dissertation aims to study a form of representation of C/C++ source code through directed graphs which will then be used by heterogeneous systems scaling algorithms (i.e PEFT, HEFT, amongst others). Towards this, an analysis of critical regions through Abstract Syntax Trees obtained from the compilation of source code with the Clang compiler, is proposed. This analysis, combined with computational models will enable the production of optimised profiles for code execution.

The programme obtained, and described in this thesis, provides a means to obtain directed graphs from compiler ASTs and identifies regions of possible parallelisation. This study of the specificity of the source code subsequently enables the use of scaling algorithms which optimises its execution in a heterogeneous architecture.

Agradecimentos

Desde logo ao Professor Jorge Barbosa pela forma notável com que conduziu a orientação desta dissertação, pela sua paciência e pelas grandes oportunidades de aprendizagem que me proporcionou.

Aos meus pais e aos meus avós, por todos os sacrifícios que fizeram, por toda a ajuda, por terem sempre colocado a minha educação acima de tudo o resto. Obrigado pelo grande exemplo que são. Obrigado.

Aos meus amigos, aos meus grandes amigos. Aos amigos que aguentaram noitadas a fazer relatórios. Aos amigos que aguentaram noitadas de gargalhadas. Aos amigos que me fizeram sentir noutro país como que na minha própria casa. Aos amigos que deixaram tudo para fazerem 2000kms.

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

AST	Árvore Sintática Abstrata
C/C++	Linguagem de Programação
DAX	Schema XML para descrição de Workflows
DOT	Graph Description Language
GCC	GNU Compiler Collection
GIT	Sistema de Controlo de Versões
HPC	High Performance Computing
IDE	Ambiente de Desenvolvimento
JSON	JavaScript Object Notation
LLVM	Infraestrutura de compilador previamente denominada Low Level Virtual Machine
UTDSP	University of Toronto Digital Signal Processing
XML	Extensible Markup Language

Capítulo 1

Introdução

Face a continua proliferação dos grandes sistemas computacionais, e à sua recente heterogeneização no que diz respeito a *hardware*, torna-se cada vez mais relevante conceber e implementar sistemas de escalonamento de trabalhos computacionais que tenham em consideração essa heterogeneidade do sistema.

Tipicamente, os processos são executados nestes sistemas computacionais numa lógica de "*first come – first served*" e com a responsabilidade de aproveitar as características do sistema colocada do lado do programador.

A alternativa proposta nesta dissertação passa por, numa primeira fase, analisar o código fonte a ser executado e, numa segunda fase, distribuir a sua execução tendo em conta as características da rede e do sistema computacional.

Esta dissertação centra-se na metodologia referente à análise do código fonte e propõe técnicas de processamento de Árvores Sintáticas Abstratas que possibilitem identificar zonas de processamento intensivo e as suas interdependências. Esta solução tira partido da Árvore Sintática Abstrata construída pelo compilador CLANG durante a fase de compilação, processa-a e constrói um grafo dirigido para análise.

Esta análise permitirá, de seguida, criar sistemas inteligentes e auto-adaptáveis, que consigam mapear aplicações a executar com os recursos disponíveis que melhor se adequem ao tipo de problema.

1.1 Contexto/Enquadramento

Esta dissertação é parte integrante do projeto Europeu Antarex. Este projeto pretende desenvolver novas técnicas de gestão de execução de aplicações em sistemas computacionais de

elevado desempenho com topologia heterogénea. Atualmente, os custos energéticos e de refrigeração são os principais entraves à expansão dos grandes sistemas computacionais. Isto é particularmente relevante nos denominados "*Exascale Level Systems*", sistemas computacionais com capacidades de processamento superiores a 1 exaFLOP (10^{18} *Floating Point Operations per Second*) onde se esperam consumos energéticos na ordem dos 60 a 130MW [1][2].

1.2 Projeto

Esta dissertação enquadra-se no contexto do Projeto Antarex [3] desenvolvido por um conjunto de instituições europeias no âmbito do programa de desenvolvimento europeu Horizon 2020[4].

Figura 1 - Logotipo Projecto Antarex

O projeto Antarex tem como objetivo estudar formas de otimizar as técnicas de gestão de recursos em sistemas computacionais de elevado desempenho. Este projeto propõe a criação de soluções que tornem os recursos computacionais auto-adaptáveis aos trabalhos a executar como forma de otimização energética.

Juntamente com esta dissertação, foi desenvolvida uma outra em paralelo intitulada "Energy-aware resource management for heterogeneous systems" [5] cujo objetivo é tirar partido dos resultados produzidos nesta dissertação (Task-Graphs) e aplicar métodos de cálculo do consumo energético em sistemas computacionais com diferentes tecidos tecnológicos, ou seja, com uma arquitetura fortemente heterogénea.

1.3 Motivação e Objetivos

Numa altura em que aumentar a eficiência do *hardware*, nomeadamente dos microprocessadores, se torna uma tarefa cada vez mais exigente [1], é fulcral que sejam tomadas medidas a nível do software de forma a minimizar, o mais possível, os custos energéticos

desnecessários, garantindo assim a expansibilidade e o aumento da capacidade dos sistemas Exascale e superiores.

Até agora, a única forma de adaptar um *software* a um sistema computacional heterogéneo, passava por fazer essa implementação de raiz, o que acarreta um custo de desenvolvimento acrescido e, além disso, não tem em consideração futuras alterações ao sistema para o qual foi desenhado originalmente.

É por isso importante, criar ferramentas de agendamento de trabalhos que, por um lado tenham em consideração clusters computacionais heterogéneos e as suas especificidades energéticas e, por outro, retirem a responsabilidade do programador de conhecer as particularidades do sistema, abstraindo-o desse trabalho acrescido.

O principal objetivo deste projeto consiste em desenvolver uma ferramenta que traduza o fluxo de execução de um programa em C/C++ segundo um grafo passível de ser futuramente analisado e distribuído para processamento num sistema computacional heterogéneo com vista a reduzir os consumos energéticos do sistema.

Mais especificamente, os objetivos deste projeto são:

- Propor formas de identificação de blocos de código com peso computacionalmente relevante
- Propor formas de representação desse código em nós de um grafo
- Desenvolver uma ferramenta que converta a AST do *Clang Parser* num grafo dirigido
- A ferramenta desenvolvida deverá produzir resultados válidos para parte significativa das estruturas de controlo de C/C++

1.4 Estrutura da Dissertação

Para além da introdução, esta dissertação contém mais 4 capítulos. No “Capítulo 2 - Representação de Aplicações Segundo Task-Graphs”, é descrito o estado da arte e são apresentados trabalhos relacionados. No “Capítulo 3 - Construção de *Task-Graph* de Programas” descrevem-se o âmbito e os detalhes da solução de forma detalhada. No “Capitulo 4 - Validação e Resultados” descreve-se como foi testada a ferramenta obtida e quais os resultados. No “Capitulo 5 - Conclusões e Trabalho Futuro”, tiram-se conclusões sobre os resultados obtidos e sugerem-se futuros desenvolvimentos desta solução.

Capítulo 2

Representação de Aplicações Segundo Task-Graphs

Neste capítulo é descrito o estado da arte e são apresentados trabalhos relacionados de forma a mostrar o que existe neste domínio e quais os problemas encontrados. Este capítulo aborda as estruturas de dados, tecnologias, compiladores e ferramentas utilizadas no desenvolvimento desta dissertação bem como estabelece comparações e paralelismos com outras abordagens semelhantes.

2.1 Introdução

Na revisão bibliográfica da área de estudo em que este projeto se centra, das tecnologias e outros estudos existentes, foi essencialmente importante observar quatro principais tópicos:

1. High Performance Computing
2. Formas utilizadas para a representação de código de fonte
3. Ferramentas de extração de conhecimento a partir do código fonte de C/C++
4. Outras ferramentas geram "task-graphs" partindo de código fonte

No tópico de High Performance Computing é essencial identificar a sua contribuição para a comunidade científica bem como as atuais tendências no que diz respeito à arquitetura destes sistemas, com uma clara tendência para o crescimento dos sistemas heterogéneos face aos tradicionais.

No que diz respeito aos pontos 2 e 3, foi importante analisar de que formas o código fonte é tradicionalmente processado e que ferramentas oferecem mais meta-informação ao programador.

O último ponto prende-se com uma análise a outras ferramentas que produzem resultados, total ou parcialmente compatíveis com o que se propõe obter neste projeto.

2.2 Computação de Elevado Desempenho

Os sistemas computacionais de elevado desempenho, ou High Performance Computing, são geralmente utilizados na resolução de grandes sistemas matemáticos como os presentes em simulações. Este tipo de sistemas tem sido implementado na resolução de problemas das mais diversas áreas: da biologia à física, passando pela meteorologia ou medicina.

Na área da biologia, um dos exemplos mais famosos da utilização de sistemas computacionais de elevada performance é o "*protein folding*" ou enovelamento de proteínas. Enovelamento de proteínas é um processo físico de transformação - enovelamento - da forma de uma proteína. De acordo com a sua composição molecular e tipo de aminoácidos que a compõem, uma proteína dobra-se e enovela-se de forma diferente. A forma resultante está associada à função dessa proteína [6][7][8]. Perceber este tipo de comportamentos de enovelamento, e os seus resultados associados, têm sido objeto de interesse da comunidade cientifica nas últimas décadas, de facto, compreender estes mecanismos moleculares oferece, ao nível da medicina, uma maior possibilidade de tratamento de doenças como a doença de Parkinson ou doença de Alzheimer.

Para calcular e simular o enovelamento das proteínas, foram desenvolvidos não só algoritmos mas também sistemas computacionais de elevado desempenho especificamente desenhados para combater este desafio como o Blue Gene pela IBM, o Molecular Dynamics GRAvity Pipe ou até o bastante noticiado projeto Folding@Home da Universidade de Stanford [7], que funciona utilizando recursos computacionais de milhões de utilizadores espalhados pelo o mundo, que contribuem para o projeto doando tempo de processamento nos seus computadores pessoais.

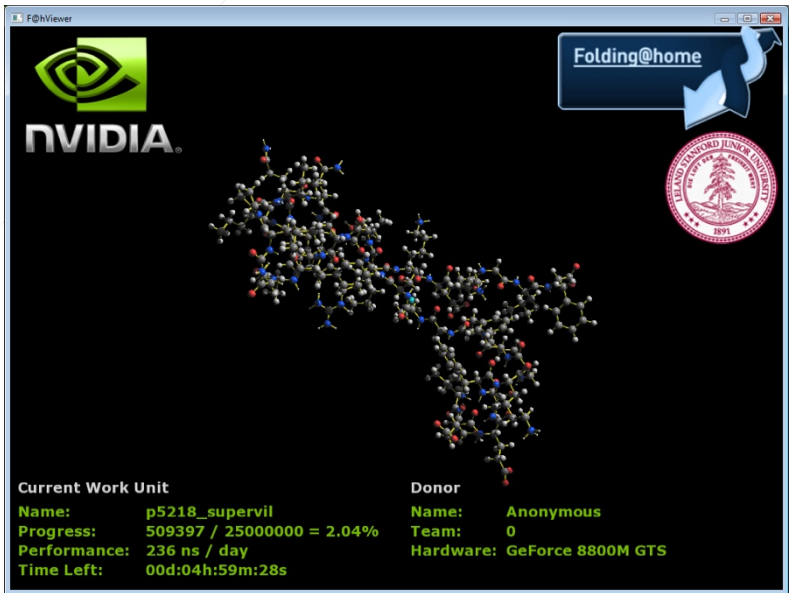

Figura 2 - Simulação de Enovelamento de Proteína no Projeto Folding@Home

Na Figura 2, um exemplo da aplicação Folding@Home a executar uma simulação do enovelamento de uma proteína. Este sistema HPC altamente distribuído é executado em computadores pessoais, cujos recursos são doados pelos seus utilizadores a este projeto. Tira sobretudo proveito dos processadores gráficos presentes nos computadores pessoais, quando estes não estão a ser utilizados.

Uma outra área cientifica que promove uma extensa utilização de High Performance Computing é a física. Na física experimental onde, por exemplo, os enormes conjuntos de dados gerados de aceleradores de partículas são analisados durante semanas, ou até meses, em alguns dos maiores supercomputadores mundiais.

Ou na Engenharia, que recorre a simulações com custos computacionais elevadíssimos, como é exemplo um dos mais recentes projetos do G8 - Grupo dos 8 países tecnologicamente mais evoluídos - de seu nome Nu-Fuse[9] que pretende desenvolver métodos de simulação de reatores de fusão nuclear, uma alternativa que, ao contrário dos reatores de fissão nuclear utilizados no último século, oferece teóricas vantagens como uma energia mais limpa e sustentável.

Tanto no primeiro exemplo, com o desenvolvimento de tratamentos para doenças que flagelam a humanidade, como num segundo exemplo em que, através da física pretende-se descobrir métodos de criação e energia limpa e sustentável está bem patente a importância para a sociedade destes sistemas, muitas vezes invisíveis.

2.3 Representação de Código

2.3.1 Árvores Sintáticas Abstratas

Uma Árvore Sintática Abstrata, é uma forma de representação de código fonte, tipicamente utilizada por compiladores.

Serve como comunicação entre o *front-end* e o *back-end* dos compiladores. O *front-end*, responsável pelas análises sintática e semântica do código fonte produz uma AST que, é então input para o *back-end* do compilador. O *back-end* utiliza essa AST para gerar código máquina destinado ao ambiente em que está introduzido.

Isto permite uma grande flexibilidade entre código fonte e sistema alvo. Por exemplo, para utilizar uma linguagem de programação já existente, numa nova arquitetura, é apenas necessário

desenvolver um novo *back-end*, desde que se utilize um front-end que produza AST's válidas. E, no sentido inverso, para codificar uma linguagem nova, numa arquitetura já existente e para a qual já existam *back-ends* disponíveis, é apenas necessário desenvolver um *front-end* de compilador para a nova linguagem, que produza uma AST que respeite o que é esperado pelo *back-end* existente [10].

Numa AST, para uma determinada expressão, um nó representa um operador, e os seus descendentes os operandos dessa operação.

A expressão $(10 \div 2) + 4$ pode ser, por exemplo, traduzida da seguinte forma:

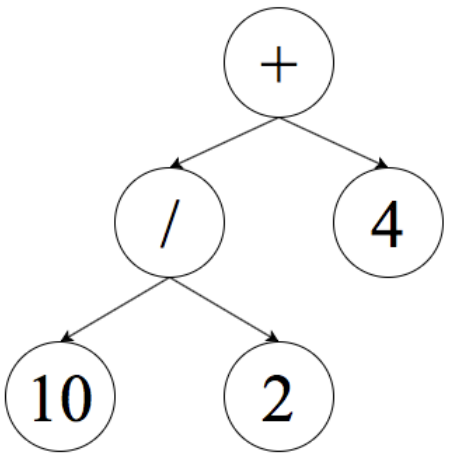

Figura 3 - AST de Expressão Aritmética

Mas esta representação não se limita a operações aritméticas, genericamente, qualquer estrutura programática pode ser representada desta forma desde que seja criado um operador para a estrutura e usados operadores que sejam componentes semanticamente importantes para essa estrutura [11].

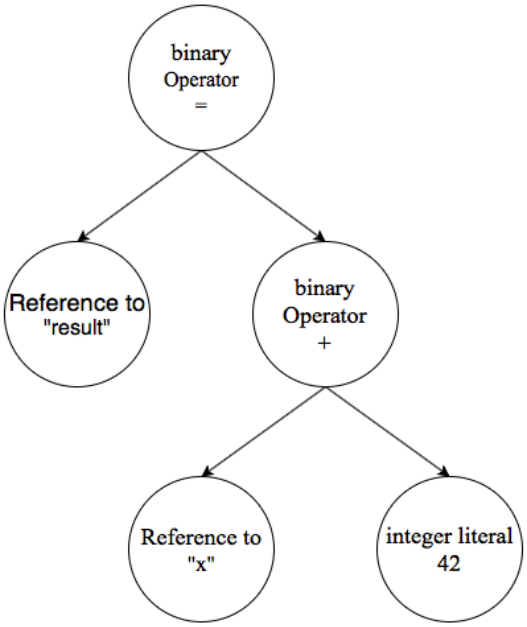

Figura 4 - AST de Atribuição a Variável

Na Figura 4, apresenta-se uma AST semelhante a produzida pelo compilador Clang, que traduz a instrução: $result = x + 42$

A diferença entre uma árvore sintática abstrata e uma árvore sintática concreta, normalmente denominada *parse tree*, é que as primeiras abstraem os pormenores sintáticos irrelevantes para uma análise semântica do código. Ou seja, enquanto que uma árvore sintática concreta, contém todos os elementos do código fonte (ex. instruções, caracteres, parêntesis, virgulas, etc.) necessários para validar sintaticamente o programa, uma árvore sintática abstrata não inclui tanta informação supérflua. Na AST não constam, por exemplo, as expressões da linguagem cujo único propósito é remover ambiguidade. A linguagem representada por uma AST pode, por isso, ser ambígua.

As vantagens das AST relativamente a árvores sintáticas concretas são [12]:

- Correspondem à gramática intuitiva da linguagem
- São mais fáceis de manipular

O segundo ponto é especialmente relevante no contexto desta dissertação, cujo objetivo passa pela manipulação e análise de código fonte. Apesar de uma análise exaustiva do código ser

teoricamente possível, é mais viável aceder a uma AST existente, proveniente de um compilador, e iterar pela sua estrutura.

2.3.2 Task-Graph

Para representar o código fonte de acordo com a sua possibilidade de ser paralelizado utilizam-se sobretudo estruturas de dados baseadas em grafos.

Estas estruturas permitem distinguir blocos de código relevantes e estabelecer as dependências entre eles. O modelo de representação *task-graph* é a forma comum de representar uma aplicação decomposta nas suas partes essenciais geralmente utilizado por aplicações de agendamento de tarefas, e por instrumentadores de código fonte que produzam *call-tree*s como é o caso do "Gprof" [13] ou do “kcachegrind” [14], por exemplo.

Esta dissertação vai seguir uma representação de *Task-Graphs* semelhante à proposta pelos criadores da ferramenta Contech [15]. Nesta representação, o *Task-Graph* possui 5 tipos de nós:

- WORK - nó de trabalho, onde fica armazenado o código da tarefa a ser executada
- CREATE-TASK - nó de criação das tarefas
- JOIN-TASK - nó de união de duas tarefas
- SYNC-TASKS - nó para sincronização de duas tarefas

- BARRIER - nó de sincronização aplicado a todas as tarefas do problema

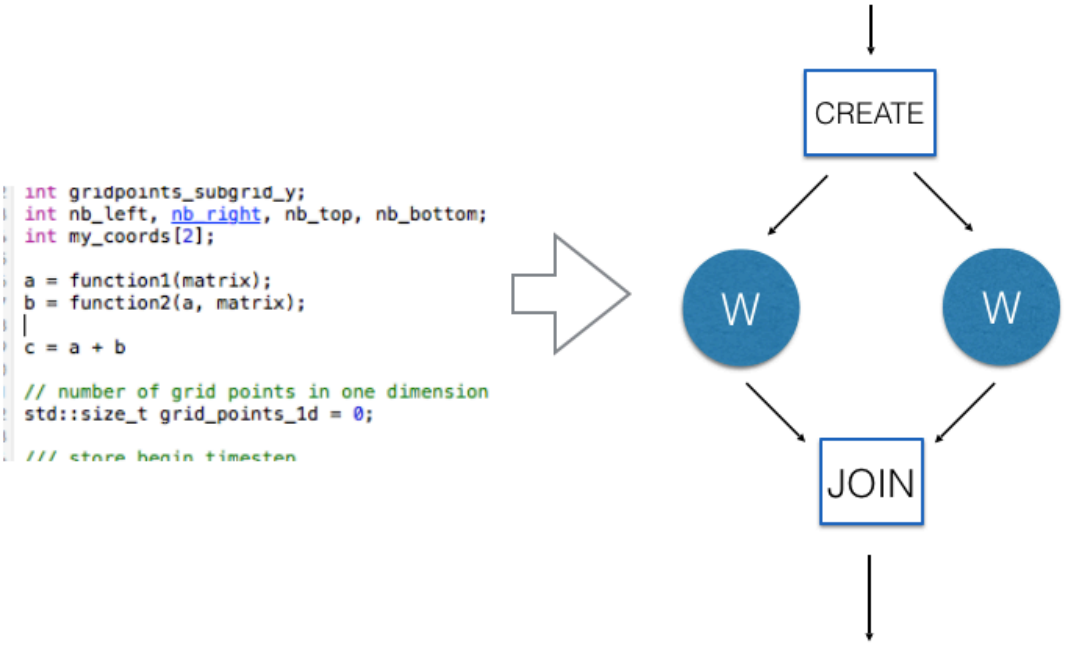

Figura 5 - Exemplo de alto nível de Task-Graph

Na Figura 5, representa-se um exemplo de alto nível de como duas funções que podem ser executadas em paralelo, ficam representadas utilizando uma estrutura *task-graph*. Neste exemplo o nó CREATE indica que se vai iniciar uma zona em que existe paralelismo. Os dois nós W (WORK) representarão, as funções "function1" e "function2" e, por último, o nó JOIN indica o fim da zona em que existe paralelismo de tarefas, e que o nó que se segue só pode ser executado quando todas as tarefas do bloco atual terminarem e agruparem os seus valores.

Os autores defendem que, estruturando o *task-graph* desta forma, transparente ao funcionamento de aplicações em paralelo, torna-se possível representar qualquer tipo de problema, seja ele em memória partilhada ou distribuída.

2.4 Analisadores Sintáticos para Código C/C++

2.4.1 Clang

Clang é um front-end de um compilador, desenvolvido pela Apple® no inicio de 2007 como alternativa ao até então utilizado GCC. Clang foi desenvolvido numa altura em que o GCC além de não responder as necessidades tecnológicas da empresa, obrigava ainda a um ciclo de desenvolvimento ineficiente e pouco acessível a novos programadores [16][17].

O Clang não foi desenvolvido com o propósito de ser um substituto ao GCC, mas sim para oferecer um conjunto diferente de funcionalidades, algumas das principais características deste compilador são:

- Árvores Sintáticas Abstratas simples de compreender e utilizar. Tal como será analisado em maior detalhe nesta dissertação, a AST produzida por este compilador é estruturada de forma descomplicada. Além de permitir uma rápida aprendizagem, torna todo o processo de desenvolvimento mais rápido e intuitivo.
- Arquitetura Baseada em Bibliotecas: permite o desenvolvimento de extensões numa lógica de programação orientada a objetos através de API's simples e extensíveis.
- Árvores Sintáticas Abstratas completas: ao contrário de outros compiladores que não fornecem acesso à AST, ou fornecem apenas acesso a uma AST simplificada, este compilador permite acesso na integra. Além disso, é possível usar o compilador para fazer *serialize* e *de-serialize* à AST e armazena-la em disco, ou ser passada por outro programa.
- Compatível com GCC: os comandos nativos de GCC são automaticamente interpretados e internamente traduzidos para a sua variante em Clang, possibilitando assim uma substituição direta entre os dois compiladores.
- Diagnósticos expressivos: mensagens de erro e avisos de compilação são expressas em maior detalhe e com maior clareza. Exemplo retirado da documentação deste compilador:

```
$ gcc-4.9 -fsyntax-only t.c
t.c: In function 'int f(int, int)':
t.c:7:39: error: invalid operands to binary + (have 'int' and 'struct A')
   return y + func(y ? ((SomeA.X + 40) + SomeA) / 42 + SomeA.X : SomeA.X);
                                       ^

$ clang -fsyntax-only t.c
t.c:7:39: error: invalid operands to binary expression ('int' and 'struct A')
  return y + func(y ? ((SomeA.X + 40) + SomeA) / 42 + SomeA.X : SomeA.X);
```

Figura 6 - Mensagens de Erro: GCC vs Clang

Neste exemplo da Figura 6, o erro do Clang permite identificar claramente qual dos operadores binários está a gerar o erro.

- Está ainda anunciada uma superior performance em termos de tempo de compilação relativamente ao GCC [18], mas com as mais recentes versões do GCC este facto já nem sempre se verifica.

O Clang oferece ainda um versátil conjunto de ferramentas ao programador para análise do código fonte. Uma característica especialmente interessante do Clang é usar uma arquitetura baseada em bibliotecas que permite facilmente identificar os seus principais componentes, e os recursos de cada um.

Alguns exemplos de bibliotecas deste *front-end* que interessam especialmente a esta dissertação são os de criação e manipulação de Árvores Sintáticas Abstratas:

1. **libast**: Representação da Árvore Sintática Abstrata de um código fonte através classes típicas ao desenvolvimento num paradigma de programação orientada a objetos.
2. **libsema**: Classe que executa a análise semântica do código fonte e gera a AST para futuro processamento.
3. **libparse**: Conjunto de classes para o processamento/análise dos componentes da AST

A AST do Clang separa o código fonte em por três classes principais: *statements*, *declarations* e *types*:

Stmt - Statements: os *statements* são a construção fulcral da AST fornecida pelo Clang, englobam as operações sobre variáveis. Por sua vez, dividem-se também em subtipos que melhor identificam cada tipo de ação. É ainda possível formar statements compostos por outros *statements*: os CompoundStmt. A Figura 7 apresenta algumas das mais relevantes subclasses de "Stmt".

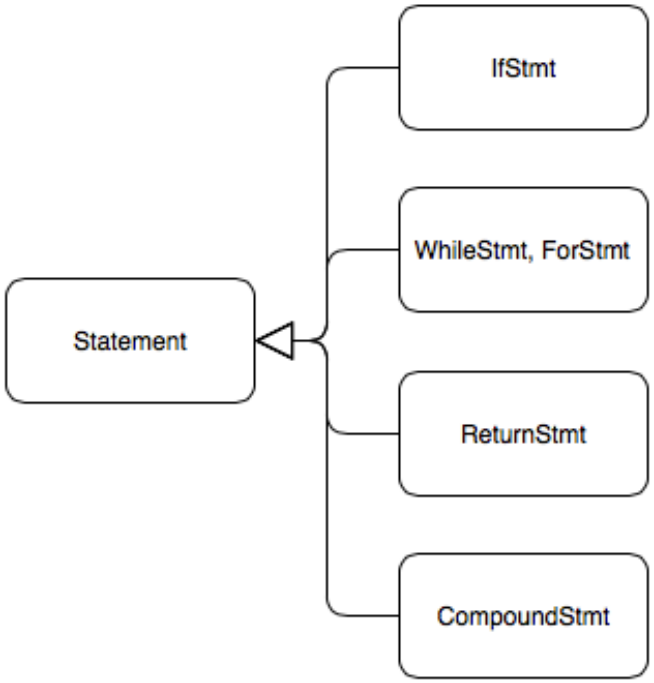

Figura 7 - Exemplos de subclasses de Stmt

Decl – Declarations: as *declarations* armazenam as instruções de definição e declaração de funções e variáveis, bem como informação do seu tipo de dados, e também argumentos e retorno no caso das funções. São ainda usadas para *Enums*. Esta classe é especialmente importante para perceber o escopo de uma declaração de uma variável, ou função.

Type – type: os types são as classes usadas para identificar os tipos de dados de variáveis bem como de retornos de funções. Tipos de dados atómicos como int ou float são associados à subclasse *BuiltinType* enquanto que tipos de dados compostos como classes ou estruturas definidas pelo utilizador são *typedefTypes* compostos por sua vez por Tipos *built-in*. Existem ainda subclasses como *ArrayType* ou *PointerType* para, respetivamente, Arrays e Apontadores.

Além dos vários tipos, é ainda possível acrescentar qualificadores a estes. Como sendo constantes, voláteis ou restritos caso na sua declaração tenham, respetivamente, as *keywords* "const", "volatile" ou "restrict", por exemplo.

A Figura 8 demonstra como o Código 1 fica estruturado no conjunto de classes da AST do Clang Parser.

```
1  int function(int x) {
2      int result;
3      result = x +  42;
4    return result;
5  }
```

Código 1 - Função de exemplo

```
`-FunctionDecl 0x102803b80 <code.cpp:1:1, line:5:1> line:1:5 function 'int (int)'
  |-ParmVarDecl 0x102803ac0 <col:14, col:18> col:18 used x 'int'
  `-CompoundStmt 0x10280d600 <col:21, line:5:1>
    |-DeclStmt 0x102803ca0 <line:2:2, col:12>
    | `-VarDecl 0x102803c40 <col:2, col:6> col:6 used result 'int'
    |-BinaryOperator 0x102803d68 <line:3:2, col:16> 'int' lvalue '='
    | |-DeclRefExpr 0x102803cb8 <col:2> 'int' lvalue Var 0x102803c40 'result' 'int'
    | `-BinaryOperator 0x102803d40 <col:11, col:16> 'int' '+'
    |   |-ImplicitCastExpr 0x102803d28 <col:11> 'int' <LValueToRValue>
    |   | `-DeclRefExpr 0x102803ce0 <col:11> 'int' lvalue ParmVar 0x102803ac0 'x' 'int'
    |   `-IntegerLiteral 0x102803d08 <col:16> 'int' 42
    `-ReturnStmt 0x102803dd0 <line:4:3, col:10>
      `-ImplicitCastExpr 0x102803db8 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x102803d90 <col:10> 'int' lvalue Var 0x102803c40 'result' 'int'
```

Figura 8 - Trecho do resultado da Clang AST

Neste exemplo, as três instruções da função "function" ficaram representadas por um "CompoundStmt" que engloba um "DeclarationStatement" com a declaração da variável "result", um “BinaryOperator” que representa a operação aritmética de soma e armazenamento do resultado em "result" e, por último um "ReturnStatement" que trata o retorno da função.

Além de objetos das classes anteriormente referidas, neste pequeno exemplo existem ainda ocorrências das classes:

- BinaryOperator: operador aritmético, neste caso de soma
- ImplicitCastExpr: forma explicita de representar o *casting* implícito que não tem representação em código
- DeclRefExpr: referencia para uma variável anteriormente declarada
- IntegerLiteral: valor constante, neste caso 42

Outros exemplos de classes de grande importância, mas que não constaram deste exemplo são:

- CompoundStmt: agrupamento de statements que permite representar instruções mais complexas

- ForStmt, WhileStmt e DoStmt: que indicam respetivamente instrução de ciclos *For*, *While* e *Do-While*
- CallExpr: representa uma chamada a uma função
- IfStmt: que indica o inicio da construção If
- UnaryOperator e CompoundAssignOperator: indicam respetivamente operações unarias e compostas. Uma operação unaria é por exemplo: a++, um exemplo de uma composta é: a += b

Para processar a Árvore Sintática Abstrata obtida, o Clang facilita três formas diferentes [19]:

- Clang Plugin
- LibTooling
- LibClang

Os *plugins* são programas de pequena dimensão que possibilitam realizar alterações na AST dinamicamente, como parte integrante do processo de compilação. Esta solução é a indicada para casos de uso em que pretende-se, por exemplo, estender o registo de avisos e erros de compilação de uma aplicação [20]

Por outro lado, o *Libtooling*, consiste numa aplicação isolada em C++ que recorre as ferramentas do Clang para processar um ou vários ficheiros de código fonte.

A última opção, *LibClang*, é a recomendada para a generalidade dos casos de uso. Esta abordagem, ao contrário das outras duas, possibilita tirar partido das funcionalidades do Clang através de outras linguagens, bem como fornece acesso a informação de mais alto nível, como por exemplo, possibilidade de percorrer a AST utilizando cursores [20].

A última opção apesar de ser teoricamente mais fácil, devido ao nível de abstração elevado, não garante ao programador acesso total à AST, o que só acontece nas duas primeiras opções. No contexto desta dissertação a opção do LibTooling é a que oferece mais flexibilidade para acesso e consulta da AST.

Neste método, define-se o código que vai ser executado em compile-time num segundo ficheiro ".cpp", após compilado executa-se este ficheiro passando como argumento o(s) ficheiro(s) a compilar e analisar, bem como os argumentos (*flags*) de compilação pretendidas.

Assim sendo, numa primeira fase, cria-se o ficheiro com o código de análise da AST a ser executado, como neste exemplo baseado num tutorial de Kevin Boos [19]:

```
1.  Rewriter rewriter;
2.  int numFunctions = 0;
3.
4.
5.  class ExampleVisitor : public RecursiveASTVisitor<ExampleVisitor> {
6.  private:
7.      ASTContext *astContext; // used for getting additional AST info
8.
9.  public:
10.     explicit ExampleVisitor(CompilerInstance *CI)
11.       : astContext(&(CI->getASTContext())) // initialize private members
12.     {
13.         rewriter.setSourceMgr(astContext->getSourceManager(), astContext->getLangOpts());
14.     }
15.
16.     virtual bool VisitFunctionDecl(FunctionDecl *func) {
17.         numFunctions++;
18.         return true;
19.     }
20. };
```

Código 2 - Libtooling Código exemplo 1/2

Em Código 2, a classe "RecursiveASTVisitor" contém métodos que são executados sempre que, no processamento recursivo da AST são executados. Estendendo esta classe, como no exemplo acima com a classe "ExampleVisitor" é possível fazer *override* a estes métodos e assim alterar o seu comportamento. Neste exemplo altera-se o comportamento da função "VisitFunctionDecl" que é chamada para cada nó "FunctionDecl", ou seja, a cada nó de declaração de uma função. Da mesma forma é possível controlar o processamento de *statements* ou *decls*, por exemplo, fazendo *override* aos métodos "VisitStmt" e "VisitDecl", respetivamente.

Baseado no código de Kevin Boos [19], este exemplo tem como simples objetivo percorrer a AST gerada pelo Clang e contar o número total de funções utilizadas. É uma tarefa simples que serve apenas para demonstração da estrutura do Clang Parser. Para invocar a classe "ExampleVisitor" com os métodos reescritos, usa-se a seguinte construção:

```
52. int main(int argc, const char **argv) {
53.     // parse the command-line args passed to your code
54.     CommonOptionsParser op(argc, argv, llvm::cl::GeneralCategory);
55.     // create a new Clang Tool instance (a LibTooling environment)
56.     ClangTool Tool(op.getCompilations(), op.getSourcePathList());
57.
58.     // run the Clang Tool, creating a new FrontendAction (explained below)
59.     int result = Tool.run(newFrontendActionFactory<ExampleFrontendAction>().get());
60.
61.     errs() << '\nFound ' << numFunctions << ' functions.\n\n';
62.     // print out the rewritten source code ('rewriter' is a global var.)
63.     rewriter.getEditBuffer(rewriter.getSourceMgr().getMainFileID()).write(errs());
64.     return result;
65. }
```

Código 3 - Libtooling Código exemplo 2/2

1. CommonOptionsParser faz *parse* aos argumentos recebidos em argv[], disponibilizando um objeto com acesso, por exemplo, ao ficheiro de código fonte recebido.
2. Cria-se um objeto da classe "ClangTool", ao qual se passa o resultado da compilação do código fonte.
3. A classe *ExampleASTConsumer*, pelo seu método "*HandleTranslationUnit*", indica que a AST deve ser processada apenas depois do parse ao código estar concluído. Para isso, é utilizado um *Visitor*.
4. A classe "*ExampleVisitor*" permite visitar qualquer tipo de nó da AST fazendo simplesmente *override* à função com o nome desse nó.
5. "*VisitFunctionDecl*" em "*ExampleVisitor*" faz *override* à função de visita de nós de declaração de funções, neste caso, apenas para somar uma unidade a uma variável global de forma a contar o número de ocorrências.

Por fim, para compilar a ferramenta de Libtooling o Clang utiliza a ferramenta Ninja [21] e, em alternativa aos tradicionais Makefiles, usa os mais poderosos scripts de CMake [22].

```
user$: llvm/build/bin/libtooling_1 test.cpp -- -Wall
```

Código 4 - Execução Ferramenta LibTooling

Em Código 4: test.cpp é o ficheiro a analisar. À sua esquerda encontra-se a execução do binário da ferramenta LibTool (neste caso chamada "libtool_1"), entre test.cpp e o separador '**--**' são argumentos de execução para o binário "libtool_1". À direita do separador '**--**' vem a lista de argumentos para o compilador Clang, neste caso de exemplo executou-se com "-Wall" para imprimir todos os avisos de compilação.

No caso de exemplo apresentado, o resultado obtido é a árvore sintática abstrata inalterada, e uma mensagem com o número de funções presentes no código.

2.4.2 GCC

Apesar dos seus quase 30 anos, o compilador GCC desenvolvido por Richard Stallman é ainda hoje um compilador extensivamente utilizado. Desde a versão 4.5, lançada em Julho de 2012 foi adicionada uma funcionalidade que permite, de forma muito semelhante aos Clang Plugins, criar extensões para o compilador. Isto permite, tal como no caso do Clang, executar alterações em tempo de compilação sem necessidade de alteração do código fonte original. Para isso, é dado acesso ao programador à AST gerada pela análise do código fonte.

Acontece, contudo, que a AST disponibilizada, é apenas uma versão reduzida e simplificada da AST gerada internamente pelo compilador. Isto deve-se sobretudo a questões políticas, sendo o GCC um *software* livre, existe um receio por parte do seu criador que, disponibilizando a AST na integra, isso possa levar a que o GCC seja utilizado integrado em *softwares* não-pagos como pré-processador [23]. Isto ficou especialmente nítido numa listagem de emails da equipa de desenvolvimento do IDE Emacs em que participou Richard Stallman. Nesta troca de mensagens, os programadores do Emacs solicitavam que fosse introduzido nas futuras versões do GCC acesso à AST integral para poderem implementar no seu IDE funções comuns a qualquer IDE moderno como *refactoring* de funções ou "auto complete" de código [24].

Este pedido foi recusado por o criador do GCC alegando receios que, outros *softwares* sem licenças de uso livre tirassem partido monetário desta funcionalidade do GCC. À data desta dissertação, esta troca de emails prolonga-se já à mais de um ano, altura em que ambas as partes estão a analisar um subconjunto de funcionalidades que possam ser implementadas com o mínimo de alterações a AST do GCC. Tendo, no entanto, já sido anunciada uma possível alteração, "*fork*", do emacs para Clang [24].

Este é apenas um exemplo de algumas das restrições impostas ao desenvolvimento de extensões por este compilador.

Contudo, este compilador tem várias outras vantagens face ao Clang, entre elas:

- Suporte para mais linguagens: além das linguagens da família C, o GCC oferece suporte também para ADA, Fortran e Java

- Suporte a mais extensões da linguagem C/C++, como por exemplo *nested functions* - funções declaradas dentro de outras funções [25].

Observando mais concretamente a sua AST é evidente que, para códigos fonte iguais a AST resultante é consideravelmente mais reduzida, e mais importante ainda, apresenta uma estrutura menos intuitiva.

Executou-se o *dump* da AST do Código 1, e obteve-se um resultado de onde consta o seguinte trecho:

```
@26     tree_list        valu: @7       chan: @34
    @27     identifier_node  strg: x        lngt: 1
    @28     identifier_node  strg: bitsizetype            lngt: 11
    @29     integer_cst      type: @21      low : 128
    @30     integer_cst      type: @21      low : 0
    @31     integer_cst      type: @21      high: -1       low : -1
    @32     convert_expr     type: @2       op 0: @35
    @33     result_decl      type: @7       scpe: @8       srcp: code.cpp:3
                        note: artificial              size: @9
                        algn: 32
@34     tree_list        valu: @2
    @35     modify_expr      type: @7       op 0: @3       op 1: @36
    @36     plus_expr        type: @7       op 0: @20      op 1: @37
@37     integer_cst      type: @7       low : 42
```

Figura 9 - Trecho do resultado da GCC AST

Na Figura 9 a linha iniciada com "@26" identifica a referencia ao argumento "x" e na linha "@34" encontra-se a representação da soma aritmética e armazenamento em memória.

É fácil observar a diferença na quantidade de informação, bem como na forma como esta informação se encontra estruturada. De facto, sem qualquer tipo de configuração adicional, o ficheiro resultado *dump* da AST do Clang ultrapassa as 1000 linhas de código, cerca de 10x mais do que o resultado do GCC, que fica aquém das 100 linhas de informação.

2.4.3 Elsa Parser

Menos conhecido que os dois anteriores, Elsa [26][27] é um *parser* para C/C++ que constrói a AST de um dado código fonte, com a interessante característica de permitir consultar a AST a cada um dos passos de compilação*: parse*, verificação de tipos e análise semântica [28].

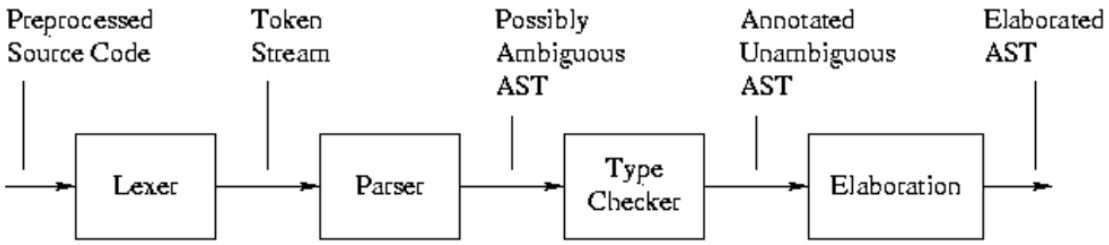

Figura 10 - Elsa Parser - Processo de geração de AST

A Figura 10 apresenta o processo de compilação/geração das ASTs. É possível consultar 3 ASTs distintas, cada uma resultante de um dos processos de compilação:

- "Possibly Ambiguous AST": Resultante do processo de *parsing*, esta AST não possui tipos, e pode, em certos casos ter ambiguidades, ou seja, não havendo certezas de algum tipo de dados são representadas as várias hipóteses para os tipos de dados que a variável possa tomar. Por exemplo, para o código "return (x) (y)" são criados dois *branches* na AST, um que assume a hipótese de se tratar de um *cast*: "return (x)y;" e outro a hipótese de se tratar de uma chamada da função x com argumento y: "return x(y);"
- Anotated Unambiguous AST: Depois de serem verificados todos os tipos, é gerada uma AST sem ambiguidades, ou seja, os casos alternativos anteriormente gerados, são agora analisados e escolhido o correto.
- Elaborated AST: AST final, além de serem processadas possíveis extensões introduzidas pelo utilizador, nesta fase apenas são detalhadas algumas funções como os construtores de classes, que não o tinham sido anteriormente.

Ao contrário dos outros dois *parsers*, que são usados como parte integrante de um compilador, o Elsa é uma ferramenta independente e isolada de geração de AST's de código C/C++, pelo que tem a desvantagem de não validar, necessariamente, a correção dos programas.

2.5 Representação de Aplicações segundo Task-Graphs

Esta secção apresenta duas das mais relevantes investigações nesta área e que tiveram como propósito a geração de Task-Graphs de aplicações.

2.5.1 Contech

Contech, desenvolvido por Brian Railing [15][29], é uma *framework* para a geração de *task-graphs* destinada à computação de elevado desempenho. Ao contrário de outras soluções não obriga a introdução de anotações no código e baseia-se em técnicas de instrumentação.

O autor utiliza o compilador Clang para obter a representação intermédia do programa, e modifica-a de forma a introduzir instruções de *profiling* que irão permitir obter métricas sobre a execução. Isto permite que o Contech seja altamente compatível, quer com todo o tipo de bibliotecas externas bem como com qualquer linguagem de programação, desde que exista um Front-End de LLVM disponível para essa linguagem.

A instrumentação funciona baseada nos acessos a memória por parte da aplicação e obriga a uma definição de funções que devem ou não ser analisadas. Para cada instrução de acesso a memória é injetada na representação intermédia uma função de recolha (instrumentação) de dados de execução e são recolhidos quatro dados: id do bloco, o tipo do evento, número de acessos a memória e o endereço de memória acedido [29].

Após esta recolha de dados da instrumentação são aplicados algoritmos de aglomeração dos resultados em tarefas "tasks", e é calculado um *task-graph*.

Alguns dos pontos mais positivos desta abordagem são:

- Compatibilidade com várias linguagens de programação, dependendo apenas da existência de um Front-end para LLVM
- Independente de anotações no código fonte original
- Fiabilidade inerente ao profiling

Algumas possíveis desvantagens desta implementação:

- O facto de as medições serem introduzidas em tão baixo nível, levanta questões relacionadas com o "probe effect", ou seja, o efeito que a obtenção de medições possa ter na alteração do resultado. De facto, esse impacto é algo que carece estudo segundo o autor[29].

2.5.2 Task graph extraction for embedded system synthesis

Numa publicação [30] de 2003, os autores Keith S. Vallerio e Niraj K. Jha apresentaram um método de tradução de AST's para *Task-Graphs* como forma de simplificação de um processo que até ao momento era feito manualmente na construção de sistemas integrados.

No caso dos *embedded systems*, ou sistemas integrados, as ferramentas utilizadas para produzir as soluções de *software*/*hardware* necessitam, assim como de outros recursos, de um *task-graph* representativo do fluxo do software. Segundo os autores, esta conversão estava sendo feita manualmente e com o risco de introdução de erros.

A solução apresentada processa a AST obtida do compilador, gera um *task-graph* com base nas dependências e dois ficheiros de código fonte: um igual ao original, obtido do compilador pelo processo habitual, e um segundo ficheiro onde foram introduzidas anotações para instrumentação em locais determinados pela estrutura do *task-graph* obtido. Ou seja, em vez de serem inseridas anotações para instrumentação a cada ponto de acesso a memória, como no caso do Contech apresentado na secção 2.5.1, nesta solução a instrumentação é de mais alto nível, inserida estrategicamente nos pontos mais relevantes.

2.6 Conclusões

Da análise do estado da arte de outros geradores de *task-graphs*, é notório o interesse por soluções como a que se propõe neste projeto. Se é verdade que tanto a solução Contech apresentada por Brian Railing [15], [29] como a proposta por Keith Vallerio et. al. [30] têm como objetivo alcançar task-graphs de aplicações C/C++, é também verdade que ambas o fazem com propósitos diferentes e de formas diferentes. A primeira solução centra-se mais na aplicação de avançadas técnicas de instrumentação do que na análise da AST, enquanto que a segunda solução faz um maior processamento da AST e usa a instrumentação como ferramenta auxiliar numa segunda fase. A solução que o projeto que esta dissertação integra, diferencia-se de ambas estas soluções uma vez que se enquadra como sendo uma solução para sistemas computacionais heterogéneos. Além disso, pelo menos nesta fase inicial, pretende-se não recorrer a instrumentação, utilizando apenas dados obtidos do compilador para a construção e enriquecimento do *task-graph*.

Da análise às ferramentas de compilação: Clang, GCC e Elsa Parser, o Clang distancia-se como sendo a única solução viável para obtenção de uma AST passível de ser interpretada. Se por um lado o GCC é claramente o compilador de C/C++ com maior quota de mercado, o Clang diferencia-se disponibilizando um conjunto de ferramentas de manipulação de dados durante as fases de compilação. O Elsa-Parser apesar de interessante não passa de um projeto meramente académico e não faria sentido construir sobre ele uma solução que poderá um dia entrar em produção.

Da investigação sobre High-Performance Computing e das "Formas de Representação de Código Fonte" pretendeu-se não só obter algum conhecimento sobre esta área de investigação, mas também explorar alguns dos casos de uso mais inovadores e relevantes, bem como perceber quais as tradicionais formas de representação de código fonte, além da forma tradicional com que qualquer programador está familiarizado.

As principais conclusões retiradas da análise ao estado da arte foram:

1. High-Performance Computing é uma área que apesar de não ser propriamente recente continua em acelerada expansão. São cada vez mais os casos de uso para estas tecnologias, bem como as soluções encontradas e desenvolvidas para obter melhores performances, e mais eficientes consumos energéticos.
2. O principal limitador da expansão dos sistemas heterogéneos não é o poder computacional, mas sim os custos/consumos energéticos.
3. O compilador Clang oferece um conjunto de meios para expandir as suas funcionalidades. É possível efetuar um processamento à AST gerada pelo compilador durante a fase de compilação.

4. A solução Contech é altamente complexa e oferece uma forma de análise de código fonte muito baseada em instrumentação de código.
5. A solução proposta por Keith Vallerio et. al [30] vai mais ao encontro do que se pretende com este projeto, contudo limita-se a gerar "task-graphs" representativos do código, sem tentar extrapolar regiões paralelizáveis.

Capítulo 3

Construção de *Task-Graph* de Programas

Este capítulo é dedicado à apresentação de detalhes de nível mais baixo relacionados com o enquadramento e implementação da solução preconizada no capítulo anterior.

3.1 Introdução

A solução proposta consiste na utilização do compilador Clang e da AST que este fornece para alimentar uma estrutura de dados própria, sobre a qual é possível executar um conjunto de filtros que alteram a sua organização, tendo como objetivo final que esta estrutura represente um grafo dirigido acíclico.

Ao longo deste capitulo será sempre usado, salvo indicação contrária, o mesmo código fonte de exemplo de forma a ser possível distinguir as transformações ao longo de cada uma das fases de processamento.

```
6  int function1(int value) {
7      int x;
8      x=2*value;
9      return x++;
10 }
11
12 int main() {
13
14     double a,b;
15     int result;
16
17     for (int i = 0 ; i < 100; i++) {
18         a++;
19     }
20
21     for (int j = 0 ; j > 20; j++) {
22         b = b * j;
23     }
24
25     if (a > b) {
26         a = a+b;
27         b= a/2;
28     } else {
29         a = b;
30     }
31
32
33     function1(a);
34
35     result = (int) (a + b);
36
37     return result ;
38 }
39
40
```

Código 5 - Código Exemplo

Este exemplo pretende ser simples e percetível, mas ao mesmo tempo cobrir os diferentes e mais relevantes casos possíveis, como a existência de ciclos paralelizáveis, uma construção condicional, uma chamada a uma função.

3.2 Âmbito da Solução

O projeto realizado, dados os constrangimentos de tempo serve como uma análise meramente académica de uma possível solução para este problema. Não sendo possível desenvolver uma solução que cubra todas as construções existentes na linguagem C/C++, implementar tratamento especifico para todas as funções das bibliotecas standard ou oferecer suporte para todos os detalhes intrínsecos a algo tão complexo como uma linguagem de programação, optou-se por, num contexto de prova de conceito, limitar o âmbito da solução a casos de teste académicos/triviais para, nesta primeira fase, determinar a fiabilidade da utilização tanto do Clang como ferramenta principal, bem como dos algoritmos de manipulação do task-graph que serão descritos nas próximas secções. Não obstante a futuros desenvolvimentos e

implementação de suporte a outras construções, o âmbito do projeto desenvolvido pode ser descrito pelos seguintes limites:

1. As construções cíclicas são consideradas a unidade mínima de processamento e são indivisíveis.
2. Numa chamada a função, variáveis que sejam passadas por referência consideram-se alteradas no escôpo dessa função. Exceto se na declaração da função esse argumento possua o modificador *const*.
3. Não são detetadas chamadas recursivas
4. Não são suportados apontadores.
5. Não são suportadas as construções: *switch*, operador ternário e blocos *try-catch*.

A limitação do ponto 1 foi introduzida, em primeiro lugar pela necessidade de gerar grafos acíclicos e, em segundo lugar, devido ao facto de serem as construções com maior tendência a gerarem um aumento da complexidade de um programa; são estas as instruções particularmente interessantes de serem paralelizadas.

As limitações 2 e 3 prendem-se também com a necessidade de eliminar grafos cíclicos em tempo útil, bem como evitar a complexidade introduzida na análise recursiva de chamadas de funções, o que originaria um problema de deteção de ciclos cuja solução não seria determinística.

As limitações 4 e 5 predem-se sobretudo com os constrangimentos temporais deste projeto, sendo limitações relativamente fáceis de serem exploradas e resolvidas em desenvolvimentos futuros.

Na análise das limitações é importante ter em conta o resultado pretendido, sendo o objetivo deste projeto obter resultados válidos optou-se por uma abordagem conservadora presente por exemplo nas limitações que dizem respeito a eliminação de ciclos e de chamadas recursivas. Se é certo que uma abordagem sem estas limitações ofereceria uma maior mais-valia para o utilizador, é também verdade que é a complexidade de conseguir resultados válidos e determinísticos é exponencialmente maior. Considerou-se que a abordagem conservadora nesta fase oferecia uma maior validade cientifica e conceptual, e que ao mesmo tempo não limitava o crescimento do projeto a medio e longo prazo.

3.3 Geração da AST

Para a geração das ASTs será usado o Parser do Compilador Clang, algumas vantagens de utilizar esta ferramenta face as outras alternativas são:

- Acesso Integral a AST
- Métodos de acesso e pesquisa na AST de fácil utilização
- Arquitetura baseada em bibliotecas que facilita o desenvolvimento

Tal como descrito em 2.4.1 existem várias formas de processar a AST obtida: usando Clang Plugins, LibTooling ou LibClang; neste projeto optou-se pelo LibTooling uma vez que além de ser mais versátil que os Clang Plugins, este método oferece acesso total à AST, o que não acontece com a ferramenta LibClang que apenas disponibiliza acessos de mais alto nível.

Mesmo que no âmbito desta solução uma abordagem utilizando LibClang pudesse ser igualmente válida, já que o Libclang continua a oferecer uma quantidade suficiente de informação sobre a AST, optou-se por uma solução em LibTooling que não é tão restritiva em termos de evolução futura.

A obtenção de informação da AST foi feita em dois âmbitos, em primeiro lugar, para obtenção de toda a informação ao nível da instrução de todo o código produzido pelo utilizador e, por outro lado, são processados todos os protótipos de funções das bibliotecas incluídas pelo utilizador.

Assim sendo, na fase de geração da AST, utilizando os métodos de visita recursiva da AST expostos em 2.4.1 – "RecursiveASTVisitor" – visitam-se 3 tipos genéricos de nós da AST:

- Declarações de funções – fazendo *overload* ao método "VisitFunctionDecl"
- Parâmetros declarados para cada função - fazendo overload ao método "VisitParmVarDecl"
- Todos os *Statements* - fazendo *overload* ao método "VisitStmt"

Da visita às declarações de funções e seus argumentos alimenta-se uma *hashtable* com a informação relativa a essa função e ao seu tipo de argumentos, se algum deles é alterado conforme as considerações do ponto 2 de 4.2 - Âmbito da Solução.

Da visita aos statements gera-se o grafo propriamente dito, dependendo do tipo especifico do *statement* visitado, diferentes processos de coleção de informação são executados como por exemplo:

- DeclRefExpr – se o *statement* for deste tipo especifico representa um nó em que é referenciada uma variável. Neste caso é importante armazenar a informação relativa a essa variável como o seu identificador, nome, tipo e tamanho.
- DeclStmt - se o *statement* for deste tipo especifico representa um nó em que é declarada uma variável. Além de armazenar a informação dessa variável este nó é imediatamente sinalizado como um nó de escrita da referida variável.

- BinaryOperator/UnaryOperator – estes nós são sinalizados como nós de escrita em variável. As variáveis em concreto serão determinadas num processamento futuro das operações executas pelos operadores.

Para o código exemplo de Código 5 o *dump* da AST é o presente no Anexo A, este *dump* é o produzido pelo clang para a execução da seguinte instrução:

```
$ clang -Xclang -ast-dump -fsyntax-only source_file.cpp
```

A seguinte imagem é uma representação gráfica de parte AST, em que cada nó contém a seguinte informação:

- Número da linha
- Tipo de Nó
- Variáveis de Leitura
- Variáveis de Escrita
- Variáveis de Acesso Indeterminado

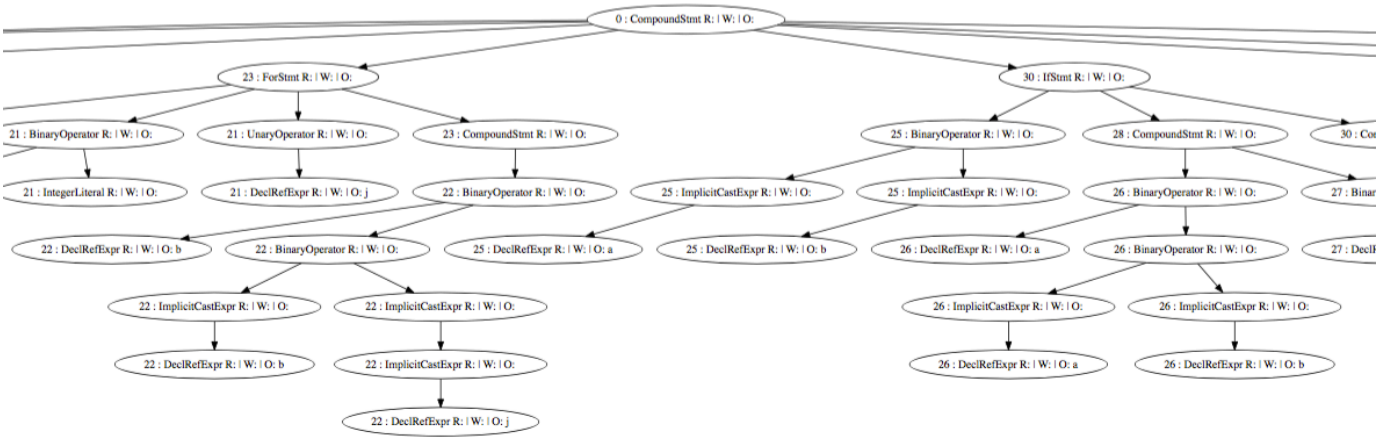

Figura 11 - AST Inicial (secção)

No total, a AST gerada resulta em 78 vértices e 77 arestas, para um código de exemplo com menos de 40 linhas de código triviais.

A secção apresentada engloba apenas as instruções das linhas 21 a 30 do código fonte original, uma instrução de ciclo “For” e uma condição “If”.

3.4 Processamento Inicial

Após a criação do grafo representativo da AST por parte do "RecursiveASTVisitor", e antes de ser possível identificar os blocos relevantes para serem paralelizados é necessário fazer um complexo processamento do grafo de modo a reduzi-lo aos nós meramente essenciais.

A solução apresentada separa este processamento inicial em 6 passos que serão agora descritos:

1. Processamento de Chamadas de Funções e Declarações
2. Processamento de Operadores
3. Processamento de Dependências
4. Corte de Ramos dispensáveis
5. Processamento de Compound Statements
6. Processamento de Condições

Só após estes passos é possível identificar que blocos é possível aglomerar e/ou paralelizar.

3.4.1 Processamento de Chamadas de Funções e Declarações

Para determinar o paralelismo e/ou aglomeração de nós do *task-graph* resultado é essencial perceber para cada referencia a uma variável, como é que essa variável é afetada pela instrução em questão, ou seja, se a instrução que referencia essa variável vai apenas executar uma leitura ou se vai possivelmente executar uma escrita.

Este primeiro passo serve para determinar o tipo de acesso a variáveis por parte de funções. Após a criação inicial da AST, obteve-se uma *hashtable* com todas as funções declaradas e seus parâmetros, neste passo cruzam-se todas as ocorrências de nós "CallExpr" – chamadas de funções - com essa informação e determina-se se as variáveis usadas como argumentos nessa chamada de função serão apenas de leitura ou também de escrita.

Neste passo são ainda encontrados todos os nós do tipo "DeclStmt" – referentes a declarações de variáveis – e as variáveis alvo de declaração são adicionas ás variáveis escritas por esse nó. Apesar de que uma declaração de variável não escreve necessariamente um valor nessa variável, não faz sentido considerar estes nós como apenas de leitura, pois também não faria nexo paralelizar um nó de declaração de variável com uma leitura.

3.4.2 Processamento de Operadores

Os operadores representam operações sobre referencias a variáveis e valores literais (constantes), tal como no passo anterior, também aqui é necessário determinar qual é o efeito da operação sobre as variáveis referenciadas: se são apenas lidas, ou também ocorre uma escrita.

O Clang define três tipos diferentes de operadores, Unários ("UnaryOperator"), Binários ("BinaryOperator") e Compostos ("CompoundAssignOperator"). Os operadores unários representam operações sobre apenas uma variável (ex.: a++, c--), e são sempre de escrita. Os operadores binários representam operações entre uma e N variáveis uma vez que é possível executar um *nesting* de operações, algo do tipo:

$$a = b + c + d + \cdots + z$$

que representa *nesting* apenas de operações binárias, sendo que também é permitido pela linguagem realizar *nesting* com operações unárias, como por exemplo:

$$a = b - - + + c$$

neste caso, o operador direito da operação binária de subtração é por sua vez uma operação unária, ou seja, ao contrário do que acontece no exemplo anterior, neste caso, uma das variáveis do membro direito faz um acesso de escrita, neste exemplo sobre a variável C.

Neste processamento são encontrados os operadores e são processados recursivamente os seus membros do lado esquerdo e direito de forma a determinarem-se todos os acessos. Por fim, dependendo do tipo de operação, classificam-se as variáveis como de escrita ou leitura.

Após a execução deste passo no processamento, todos os nós que acedem a variáveis têm identificadas as variáveis a que acedem apenas para leitura e aquelas que modificam.

3.4.3 Processamento de Dependências

Estando todos os acessos identificados e classificados este passo no processamento propaga esses acessos para os pais dos nós que fazem o acesso. Até ao momento, os nós que realizam acessos limitam-se a Operadores e Chamadas de Funções, é necessário para o processamento que se segue, que essa informação esteja ao nível dos "CompoundStaments" para que seja possível classificar blocos maiores quanto aos seus acessos. Um exemplo disso é o caso dos ciclos, sendo estes uma unidade indivisível no contexto deste projeto, é necessário saber ao nível do nó "ForStmt" quais são as variáveis a que este acede.

Após este passo, todos os "CompoundStmt", "ForStmt", "WhileStmt", entre outros, têm informação sobre os acessos dos seus nós filhos.

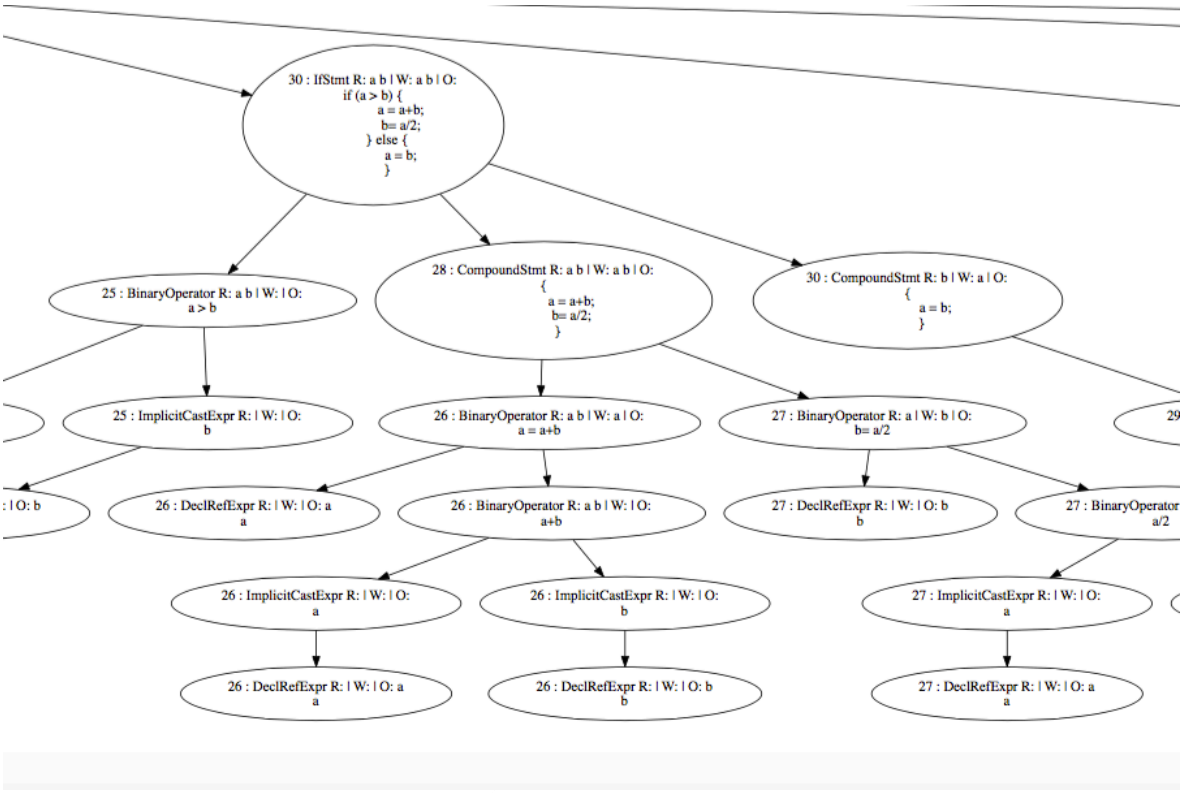

Figura 12 - Grafo Exemplo - Proc. Dependências (secção)

Na Figura 12 mostra-se uma secção da do grafo após o cálculo de dependências, é possível ver que na descrição dos nós, existem dados após as letras R (variáveis de leitura), W (variáveis de escrita).

3.4.4 Corte de Ramos dispensáveis

Até agora todo o processamento limitou-se a determinar os acessos a variáveis e a determinar dependências, este é o primeiro passo que vai executar alterações mais profundas no grafo. De forma a eliminar nós sem relevância para o resultado final este processo itera sobre os nós e remove todos aqueles que não tenham como nó-pai um “CompoundStmt” ou um “IfStmt”. É desta forma que se reduz sequencias de operações complexas a um conjunto de nós mais reduzido e fácil de tratar. Isto permite, por exemplo, reduzir operações *nested* a apenas uma operação, eliminar os nós de casting, e reduzir ciclos a apenas um nó.

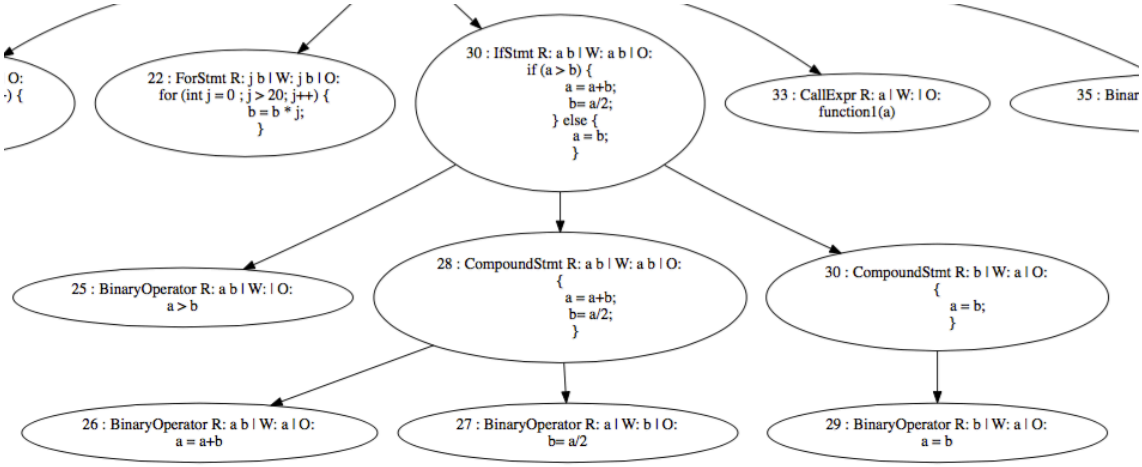

Figura 13 - Grafo Exemplo - Corte Ramos (secção)

Comparando a Figura 12 com a Figura 13 é possível identificar alguns dos nós que foram removidos, e simplificações que foram feitas.

3.4.5 Processamento de Compound Statements

Este é o passo de processamento que mais transforma o aspeto do grafo, o que este passo efetivamente faz é reduzir o número de filhos de cada nó a apenas um (salvo os casos de condições). Isto permite fazer uma representação sequencial das instruções.

A transformação é feita para cada Compound Statement, removendo as ligações para todos os seus filhos exceto o primeiro e depois ligando-os entre eles, mantendo a mesma ordem: o primeiro filho liga com o segundo, o segundo com o terceiro, etc. O último descendente do Compound Statement passa a conectar-se com o próximo nó irmão (com o mesmo nível na árvore) do Compound Statement.

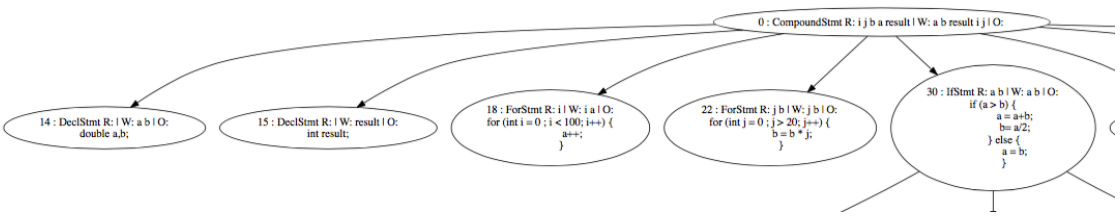

Figura 14 - Grafo Exemplo - Proc. Compound – Antes (secção)

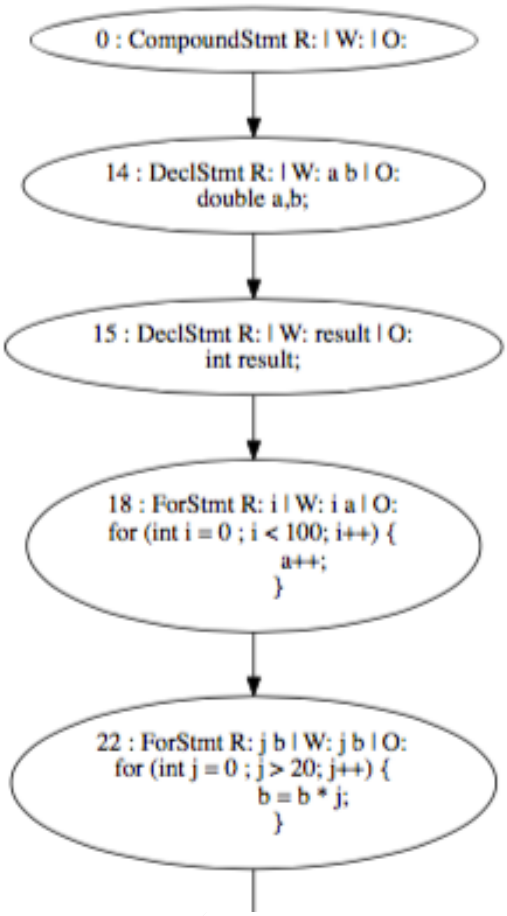

Figura 15 - Grafo Exemplo - Proc. Compound – Depois (secção)

A Figura 14 e a Figura 15 mostram exatamente os mesmos nós, antes e depois da execução deste processamento.

3.4.6 Processamento de Condições

As condições (*If Statements*) são um caso especial e não podem ser tratados como no passo de processamento anterior. Tipicamente os nós IF têm três nós:

1. Condição – tipicamente um "BinaryOperator"
2. Código se condição verdadeira – representado por um "Compound Statement"
3. Código se condição for falsa – também representado por um "Compound Statement"

Assim sendo, não faz sentido tratar da mesma forma que se trataram os "Compound Statements": ligando cada filho ao seguinte, e o último filho à próxima instrução. Isto originaria uma representação incorreta em que o código a executar se a condição fosse falsa seria executado

sempre após o código do caso verdadeiro. Além disso, se se efetuasse um processamento como no passo anterior, ao paralelizar e aglomerar blocos, gerar-se-iam grafos resultado inválidos. Existem duas formas para tratar este problema:

- Desdobrar cada possível resultado de um IF em dois IFs, um deles negado.
- Manter os dois ramos e forçar sincronização no fim dos dois ramos.

A primeira alternativa consiste em fazer uma transformação ao código fonte, essencialmente substituindo cada IF por dois, para executar cada um dos ramos possíveis. Sendo que a condição do segundo IF é precedida pelo operador negação.

A segunda alternativa permite não alterar o código, sendo apenas uma alteração ao nível da representação interna no grafo. E obriga apenas a adicionar um novo nó no final do IF para distinguir este bloco de um bloco de paralelismo.

No Código 6 apresenta-se resultado da transformação do código fonte para a primeira alternativa:

```
32      if (a > b) {
33          a = a+b;
34          b= a/2;
35      }
36
37      if !((a > b)) {
38          a = b;
39      }
40
```

Código 6 - Proc. Condições - 1ª Alternativa

Na imagem abaixo apresenta-se o resultado do processamento segundo a segunda alternativa. Neste caso o nó IF tem 3 linhas de descendentes, a primeira representa a condição, a segunda o código do caso verdadeiro, e a última, o código do caso falso.

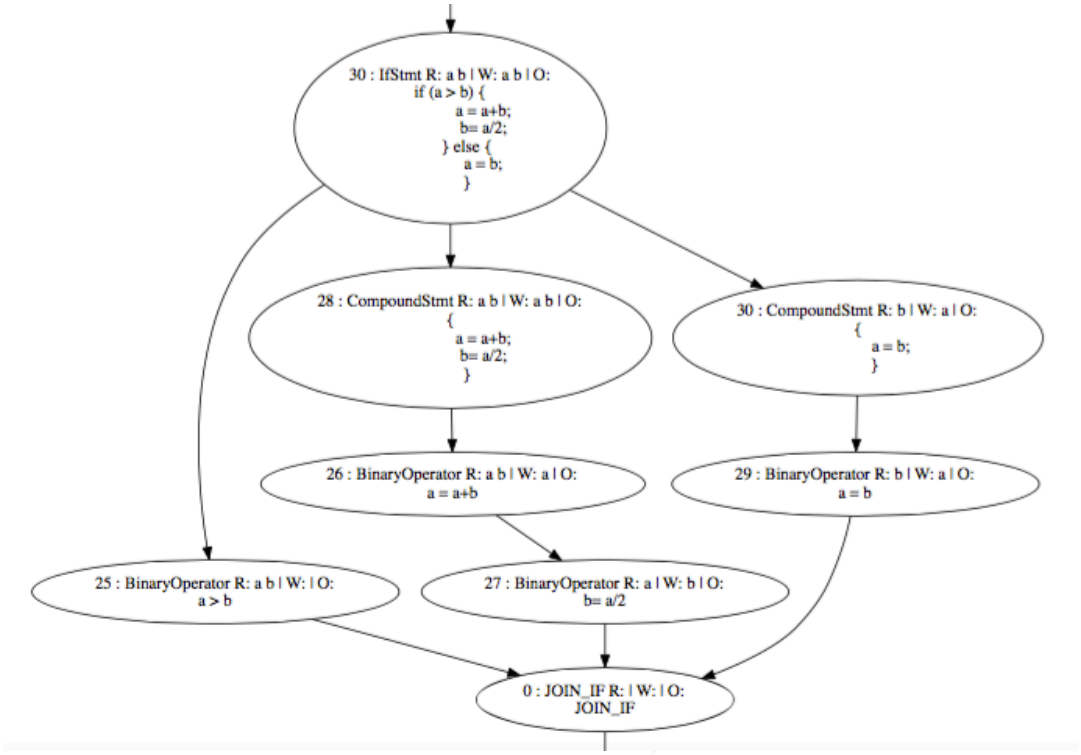

Figura 16 - Proc. Condições - 2ª Alternativa (secção)

Optou-se pela segunda implementação uma vez que oferecia os mesmos resultados sem obrigar a um manuseamento mais complexo do código fonte.

Após algum processamento e remoção de nós redundantes, bem como do ramo do grafo dedicado à condição, no final deste processo o resultado obtido é o apresentado na Figura 17.

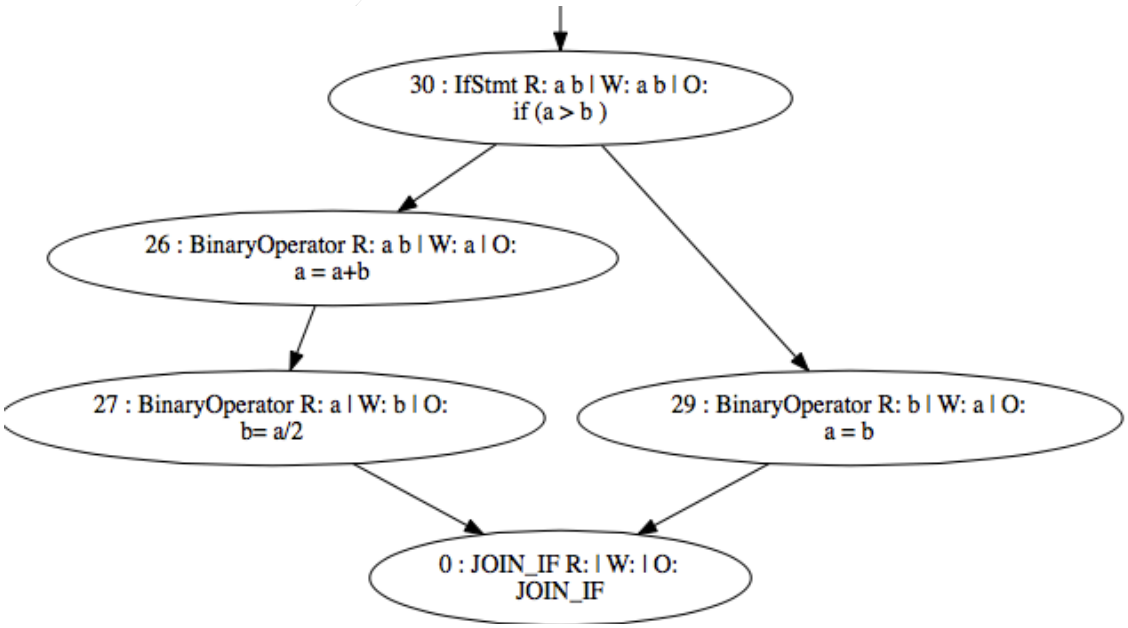

Figura 17 - Proc. Condições - Resultado Final (secção)

3.4.7 Conclusão

Em suma, após estas seis etapas de processamento o grafo obtido, para o exemplo inicial é o apresentado na seguinte figura:

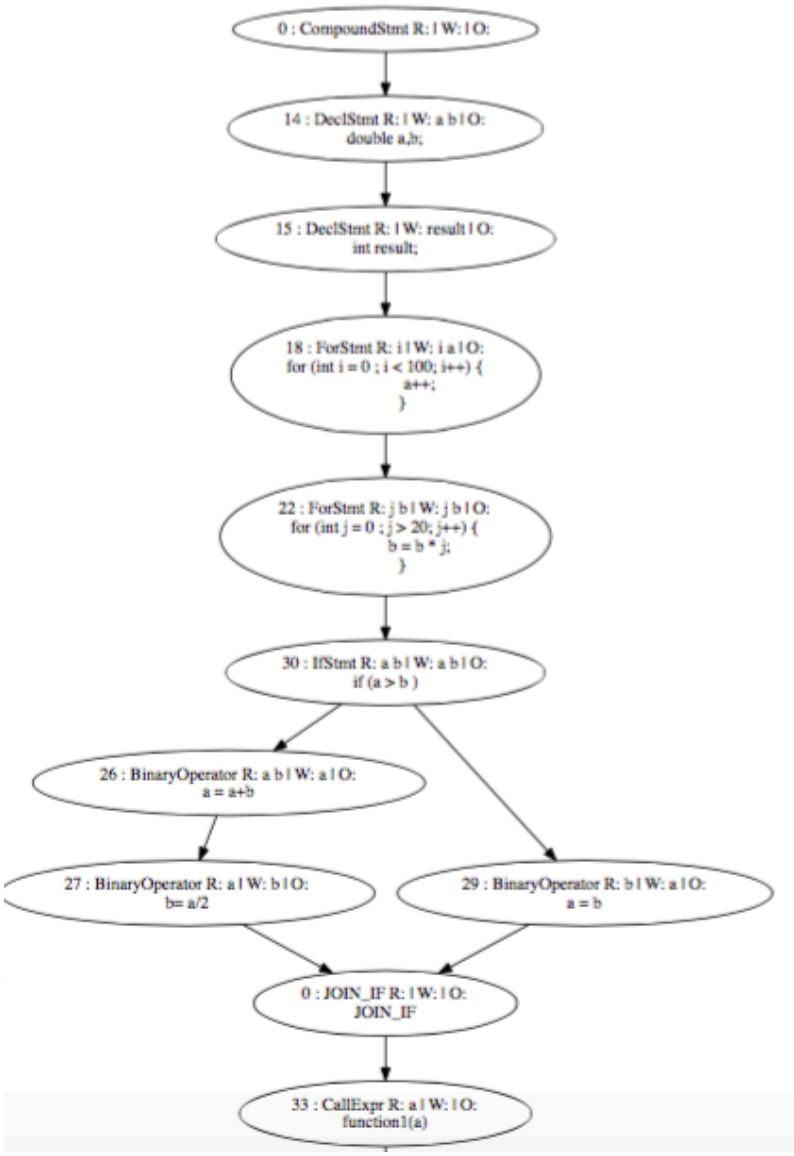

Figura 18 - Grafo após processamento inicial

Esta representação demostra o caráter sequencial do código fonte e permitirá agora proceder a alterações mais complexas como são a aglomeração de blocos e/ou identificação de regiões paralelizáveis.

3.5 Aglomeração de Blocos

Este passo prende-se com a necessidade de reduzir a granularidade dos blocos a paralelizar. Como por vezes o custo da paralelização é superior ao custo de processamento de um bloco, especialmente se este bloco for de reduzida complexidade, faz sentido tentar aglomerar instruções e criar blocos de maior complexidade quando possível. Com esse objetivo desenvolveu-se este passo do processamento.

Este processo é trivial e consiste e percorrer o grafo da raiz até ao último nó e, a cada nó, perceber se é possível fazer uma união com o nó seguinte. Para que tal seja possível, existem um conjunto de restrições, que facilmente poderão ser estendidas e complementadas.

É possível unir dois nós nas seguintes condições:

1. Existe uma relação de parentesco entre os dois nós
2. Nenhum dos nós é do tipo JOIN ou JOIN_IF
3. Nenhum dos nós é do tipo “For”, “While” ou “DoWhile”
4. Nenhum dos nós é uma chamada de função
5. Nenhum dos nós é do tipo IF (condição inicial)

O ponto 1, é trivial e evitar aglomerações de blocos que invalidassem a estrutura do programa. Se assumirmos 3 nós, A, B, C em que B é filho de A e C filho de B; só é possível A e C serem aglomerados, se A e B forem aglomerados primeiro.

O ponto 2 impede de unir blocos com pontos de controlo de fluxo.

Os pontos 3 e 4 assumem as considerações tomadas no capitulo 3.2 e que todos os tipos de ciclos são a unidade mínima e não devem ser unidos tal como não devem ser paralelizados.

O ponto 5 obriga a que a condição do bloco IF seja um bloco separado. Ao contrário dos outros pontos, este não é algo forçado pela necessidade de ter um grafo válido, tendo sido apenas uma decisão de desenvolvimento baseada no facto de, ao não aglomerar este nó facilita-se a identificação deste bloco.

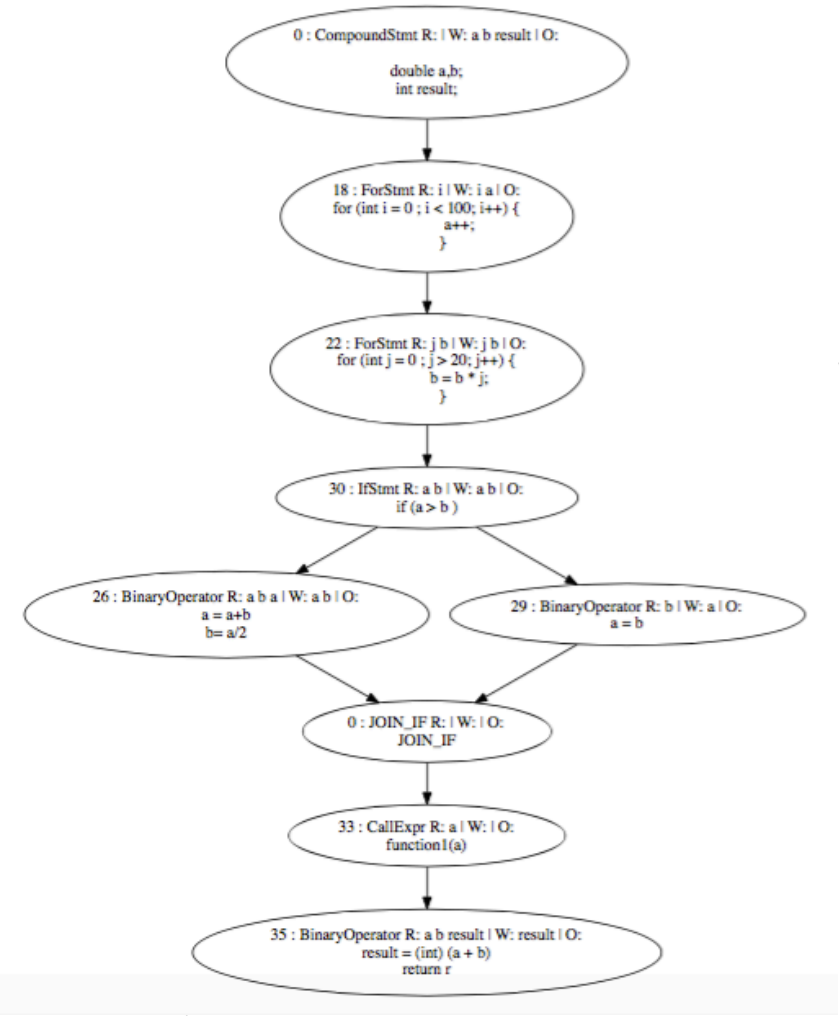

Figura 19 - Grafo após agregação

Na Figura 18 apresenta-se também o o grafo antes de ver os blocos aglomerados, e na Figura 19, após este processamento.

3.6 Criação de pontos de *Fork* e *Join*

A deteção de pontos de *fork* and *join* foi um dos processos mais complexos desenvolvidos nesta solução. O algoritmo é executado do último nó do grafo até ao primeiro e, essencialmente, para cada dependência de cada nó, pesquisa a última dependência da variável. Se existir alguma dependência num nó acima do atual, que não seja o pai é desencadeado um processo de introdução de dois nós de controlo:

- Nó FORK após a última dependência encontrada.

- No JOIN antes do nó atual para forçar a sincronização

Este algoritmo não oferece sempre uma solução ótima, uma vez que não tem em consideração o peso computacional de cada nó. Contudo, pretende oferecer soluções sempre válidas no âmbito do problema, ou seja, o código fonte após as transformações deve dar um resultado válido.

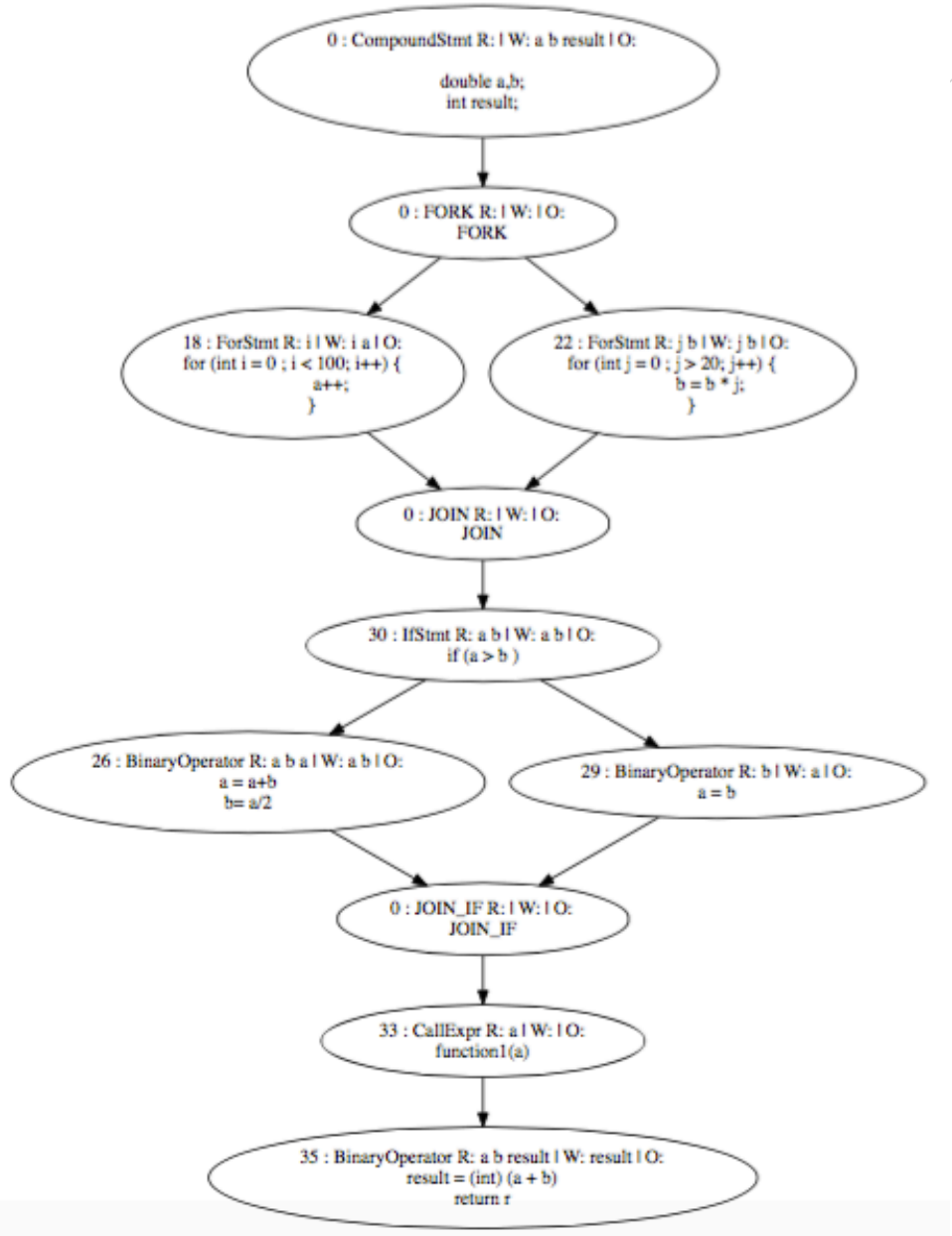

Figura 20 - Grafo após separação

Na Figura 20 apresenta-se o resultado de aplicar o algoritmo de identificação de dependências no exemplo anterior.

3.7 Geração de Output

A solução apresentada gera dois tipos de output: em primeiro lugar um output do tipo DOT apenas para permitir visualizar graficamente o resultado e em segundo lugar gera um output para facilitar o processamento e análise do grafo da fase seguinte.

Quanto ao output do grafo para análise, utiliza-se a representação DAX utilizada pelo SIMGRID [31][32] que é, por sua vez, baseada numa representação estabelecida pela ferramenta de representação de *workflows* Pegasus [33].

A representação DAX é um *schema* de XML que estabelece um conjunto de regras para a representação de estruturas tipo grafo em XML.

De forma a estender as funcionalidades do ficheiro DAX, é ainda gerado um terceiro ficheiro no formato JSON que é processado juntamente com o XML e que contém informação extra sobre o grafo impossível de adicionar ao DAX/XML.

3.8 Ferramentas Utilizadas

As ferramentas utilizadas na implementação deste projeto foram:

- Clang versão 3.9.0
- CMake versão 3.5.20160209
- Aplicação GraphViz versão 2.36 para a visualização dos grafos através dos ficheiros DOT (http://www.graphviz.org)
- Biblioteca "JSON for Modern C++" implementada por Niels Lohmann (https://github.com/nlohmann) para a geração do ficheiro JSON
- IDE e GIT

Validação e Resultados

4.1 Metodologia

De forma a validar os resultados obtidos, recorreu-se à execução da ferramenta num conjunto de programas de teste de baixa e média complexidade. Além de um conjunto de 20 códigos de teste desenvolvidos para testar casos específicos, recorreu-se a 12 *benchmarks* mais complexos incluídos no UTDSP Benchmark Suite [34].

Os códigos de teste usados contêm vários casos desde os mais triviais a outros possivelmente problemáticos, como por exemplo:

- *Inputs*/*Outputs*
- Utilização de funções bibliotecas *standard* da linguagem
- Funções criadas pelo utilizador com argumentos por referencia com e sem modificador *const*.
- Manipulação de *Arrays* e Matrizes
- Sequencias de Ciclos
- *Nesting* de Ciclos
- Encadeamento de operadores
- Operadores binários

O UTDSP Benchmark Suite contém um total de 18 *benchmarks* distribuídos entre *kernels* e aplicações, foram selecionados os 12 exemplos que cumpriam as restrições ao âmbito do problema como descritas no capitulo 3.2. Os 12 programas de *benchmark* utilizados foram:

- FFT – Fast Fourier Transform
- FIR – Finite Impulse Response filter
- IIR – Infinite Impulse Response filter
- LATNRM – 32nd-Order Normalized Lattice filter
- LMSFIR - Least mean squares
- MULT – Multiplicação de Matrizes
- ADPCM - Adaptive Differential Pulse-Code Modulation

- Edge Detect – Deteção de arestas numa imagem com 256 níveis de cinzento com 128 por 128 pixéis de resolução.
- Histogram – Melhoria numa imagem a 256 níveis de cinzento por aplicação do método "Global Histogram Equalization"
- LPC – Implementação de um codificador "Linear Predictive Coding"
- Compress – Compressão de uma imagem por um factor de 4:1
- Spectral Estimation – Cálculo da intensidade espectral de um trecho de voz utilizando médias de periodogramas

A utilização dos programas em *benchmark* pretende essencialmente comprovar o suporte à linguagem, tal como feito por Keith Vallerio et al. [30].

O conjunto de testes próprios foi o resultado do desenvolvimento e pretende testar um conjunto de outros casos limite bem como permite fazer uma melhor análise do desempenho do método de *Fork* e *Join* (Capitulo 3.6).

Foram executados os 32 códigos de exemplo, e os resultados foram classificados quanto aos seguintes parâmetros:

1. Obtenção de Resultado – foi obtido um resultado pela execução da ferramenta
2. Validade do Resultado – o resultado obtido é um grafo válido
3. Qualidade da Solução – classifica a qualidade dos pontos de *fork*/*join* introduzidos
4. Tempo de Compilação – tempo demorado na compilação do código de teste

Consideram-se válidos no contexto desta dissertação grafos com as seguintes características:

a) Não apresentam ciclos
b) Os únicos nós com mais do que um nó filho são os nós de IF e FORK
c) Os únicos nós com mais do que um pai são os nós de JOIN_IF (término de bloco condicional) e JOIN (término de bloco paralelo).

A qualidade da solução é uma medida subjetiva, obtida pela análise a posição dos pontos de FORK e JOIN bem como da sua quantidade. As soluções serão classificadas como Fracas (F), Razoáveis (R) ou Boas (B).

4.2 Resultados

4.2.1 Testes Próprios

Caso de Teste	*Resultado Obtido*	*Resultado Válido*	*Qualidade da Solução*
#0 – Double Loop	S	S	B
#1 – Conditional + Loop	S	S	R
#2 – User Functions	S	S	B
#3 – Pointers	S	S	R
#4 – Pass By Reference	S	S	B
#5 – Nested Loops Diferent type	S	S	B
#6 – Nested Loops + Functions	S	S	B
#7 - Conditional + Functions	S	S	B
#8 - Conditional + Functions	S	S	B
#9 – Included Libraries	S	S	B
#10 - Included Libraries 2	S	S	B
#11 – Compound Assignment	S	S	B
#12 – Do-While Loop	S	S	R
#13 – Loop Paralelization	S	S	B
#14 - Input/Output	S	S	B
#15 - Input/Output	S	S	B
#16 - Included Libraries 3	S	S	B
#17 - Included Libraries 4	S	S	R
#18 - Included Libraries 5	S	S	R
#19 - Included Libraries 6	S	S	B

Tabela 1 - Resultados Testes Próprios

4.2.2 UTDSP Benchmark

Caso de Teste	*Resultado Obtido*	*Resultado Válido*	*Qualidade da Solução*
FFT	S	N	F
FIR	S	S	B
IIR	S	S	B
LATNRM	S	S	R
LMSFIR	S	S	B
MULT	S	S	B
ADPCM	S	S	R
Edge Detect	S	S	B
Histogram	S	S	B
LPC	S	S	R
Compress	S	S	R
Spectral Estimation	S	S	B

Tabela 2 - Resultados Benchmarks UTDSP

4.2.3 Outros resultados

Em média os tempos de execução dos casos de teste próprios ficaram abaixo de 0.4 segundos, nos programas da *benchmark* a média foi de 0.5 segundos.

4.3 Análise

Apesar do volume de casos de teste ser de pequena dimensão, considera-se apropriado para o contexto do problema e de acordo com os limites estabelecidos para esta fase inicial do projeto.

Dos 32 casos de teste todos produziram uma solução, e apenas um dos resultados obtidos não cumpriu os critérios de validez. Neste caso em concreto, o algoritmo de identificação de "FORKs" e "JOINs" gerou um nó de "JOIN" inválido que poderia levar a ocorrência de um ciclo.

De resto, 58% dos programas da UTDSP testados as soluções foram classificadas como boas, nos testes próprios este valor sobe para 75%. Esta diferença deve-se sobretudo ao facto

destes testes terem sido utilizados durante todo o desenvolvimento do projeto numa metodologia de *Test-Driven Development*. Contudo, os resultados obtidos da execução da suite UTDSP, apesar de tímidos, permitem confirmar e validar os resultados obtidos nos testes próprios.

Capítulo 5

Conclusões e Trabalho Futuro

5.1 Satisfação dos Objetivos

Neste trabalho planeou-se e desenvolveu-se uma ferramenta que permite obter *Task-Graphs* de código fonte C/C++ através da interpretação e manipulação da Árvore Sintática Abstrata do compilador Clang. Estes Task-Graphs apresentam regiões onde é possível a utilização de técnicas de paralelismo e servem como *input* para uma outra ferramenta, desenvolvida em paralelo e no contexto do mesmo projeto, que permite gerar estimativas dos custos energéticos para a execução do código fonte em vários tipos de sistemas computacionais.

Apesar da ferramenta ter algumas limitações conceptuais devidas ao âmbito em que foi desenvolvida, quando testada em 12 dos 18 *benchmarks* da UTDSP[34] obteve resultados válidos para 11 desses *benchmarks*. Pelo que surge como uma base sólida para futuras iterações de desenvolvimento.

Os objetivos iniciais eram ambiciosos e na sua maioria não saíram defraudados. A complexidade do compilador Clang foi o maior entrave ao desenvolvimento, mas, no fim produziu-se uma ferramenta sólida, que apresenta resultados interessa resultados interessantes dentro do seu âmbito de solução e, não menos importante, estabelece uma boa base para futuras iterações e desenvolvimento.

5.2 Trabalho Futuro

Esta dissertação apresenta um resultado positivo, contudo, o âmbito da solução é algo limitado e, dá aso a futuras extensões e desenvolvimentos.

Em primeiro lugar seria crucial estender o suporte da ferramenta desenvolvida às restantes construções das linguagens C/C++ tais como: *switch-cases,* blocos *try-catch,* operador ternário, etc.

Em segundo lugar, seria interessante adicionar um módulo para suporte de *pragmas* definidos pelo utilizador. Estas anotações seriam úteis para fornecer informações extra sobre os blocos computacionais a serem analisados, podendo também servir como limitadores de blocos.

Seria também pertinente analisar outras possíveis formas de tratamento de ciclos, uma vez que a solução apresentada nesta dissertação pode por vezes criar uma abstração demasiado grande para programas que recorram a muito *nesting* de ciclos. Seria interessante desenvolver outras formas de análise destas construções que permitam ter mais detalhe destes blocos sem que sejam introduzidos ciclos no grafo resultado.

Em quarto lugar, mas não menos importante, seria essencial continuar o desenvolvimento dos algoritmos usados para a aglomeração e paralelização de nós do grafo (criação de pontos FORK e JOIN). Apesar de bastante complexo, algoritmo apresentado tem ainda algumas falhas como apresentadas em 4.3, além disso, é um algoritmo que toma decisões sobre onde introduzir paralelismo baseado apenas no tipo de nós. Seria interessante complementar o algoritmo de forma a ter em contra outra informação que pudesse ser obtida por *profiling*.

Por último, seria interessante analisar uma possível expansão desta solução de forma a contemplar outras linguagens de programação como ObjectiveC ou Java, já suportadas pelo compilador Clang.

Referências

[1] S. Kamil, J. Shalf, and E. Strohmaier, "Power efficiency in high performance computing," *2008 IEEE Int. Symp. Parallel Distrib. Process.*, pp. 1–8, 2008.

[2] H. Simon, "Modeling and Simulation at the Exascale for the Energy and the Environment," *Report*, pp. 1–174, 2008.

[3] "Antarex." [Online]. Available: http://www.antarex-project.eu. [Accessed: 20-Jun-2016].

[4] "Horizon 2020." [Online]. Available: https://ec.europa.eu/programmes/horizon2020/. [Accessed: 20-Jun-2016].

[5] E. Fernandes, "Energy-aware resource management for heterogeneous systems," Universidade do Porto, 2016.

[6] K. Y. Sanbonmatsu and C.-S. Tung, "High performance computing in biology: multimillion atom simulations of nanoscale systems.," *J. Struct. Biol.*, vol. 157, no. 3, pp. 470–80, Mar. 2007.

[7] D. A. Bader, "Computational biology and high-performance computing," *Commun. ACM*, vol. 47, no. 11, p. 34, Nov. 2004.

[8] K. Dill, "The protein folding problem: a major conundrum of science," *Presented at TEDxSBU, Stony Brook University*, 2013. [Online]. Available: https://www.youtube.com/watch?v=zm-3kovWpNQ. [Accessed: 10-Feb-2016].

[9] G8 Research Councils Initiative on Multilateral Research Funding, "The New Fuse Project." [Online]. Available: http://www.nu-fuse.com. [Accessed: 10-Feb-2015].

[10] J. Lucas, "Thoughts on the Visual C++ Abstract Syntax Tree (AST)," 2006. [Online]. Available: https://blogs.msdn.microsoft.com/vcblog/2006/08/16/thoughts-on-the-visual-c-abstract-syntax-tree-ast/. [Accessed: 10-Feb-2016].

[11] M. Lam, R. Sethi, J. Ullman, and A. Aho, *Compilers: Principles, techniques, and tools*. 1986.

[12] J. Cardoso, "Slides da Unidade Curricular de Compiladores - Análise Lexical, Sintática e Semântica," 2014.

[13] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof," *ACM SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, Jun. 1982.

[14] J. Weidendorfer, "kcachegrind, call graph viewer." [Online]. Available: https://kcachegrind.github.io/html/Home.html. [Accessed: 12-Feb-2015].

[15] B. P. Railing, E. R. Hein, and T. M. Conte, "Contech: Efficiently Generating Dynamic Task Graphs for Arbitrary Parallel Programs," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, pp. 1–24, 2015.

[16] S. Naroff, "New LLVM C Front-end," in *LLVM Developers' Meeting Proceedings*, 2007.

[17] LLVM Developer Group, “clang: a C language family frontend for LLVM.” [Online]. Available: http://clang.llvm.org.

[18] C. Lattner, “LLVM 2.0 and Beyond!,” *Google Tech Talk, Mountain View, CA*, 2007. .

[19] K. Boos, “Clang Tutorial Part I: Introduction.,” *Bits Bytes Boos*. .

[20] LLVM Developer Group, “Clang 3.9 documentation.” [Online]. Available: http://clang.llvm.org/docs/Tooling.html. [Accessed: 01-Feb-2016].

[21] E. Martin, “Ninja, a small build system with a focus on speed.” .

[22] A. Cedilnik, B. Hoffman, B. King, K. Martin, and A. Neundorf, “CMake - Cross Platform Make.” [Online]. Available: https://cmake.org. [Accessed: 10-Feb-2016].

[23] “Lead Emacs Developer considering Forking over GCC and AST Issues,” *Hacker News*.

[24] Emacs Mailing Lists Archive, “Re: Emacs contributions, C and Lisp,” 2015. .

[25] GCC online documentation, “Nested Functions - Using the GNU Compiler Collection (GCC).” .

[26] S. McPeak and D. Wilkerson, “Elsa: The Elkhound-based C/C++ Parser.” 2005.

[27] S. McPeak and G. C. Necula, “Elkhound: A fast, practical GLR parser generator,” *Compil. Constr. Proc.*, vol. 2985, Jan. 2004.

[28] C. Wagner, T. Margaria, and H.-G. Pagendarm, “Comparative Analysis of Tools for Automated Software Re-engineering Purposes,” in *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, 2006, pp. 433–440.

[29] B. P. Railing, “Collecting and representing parallel programs with high performance instrumentation,” 2015.

[30] K. S. Vallerio and N. K. Jha, “Task graph extraction for embedded system synthesis,” in *16th International Conference on VLSI Design, 2003. Proceedings.*, 2003, pp. 480–486.

[31] H. Casanova, A. Legrand, and M. Quinson, “SimGrid: A Generic Framework for Large-Scale Distributed Experiments,” in *Tenth International Conference on Computer Modeling and Simulation (uksim 2008)*, 2008, pp. 126–131.

[32] H. Casanova, “Simgrid: a toolkit for the simulation of application scheduling,” in *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2001, pp. 430–437.

[33] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny, “Pegasus: Mapping Scientific Workflows onto the Grid,” Springer Berlin Heidelberg, 2004, pp. 11–20.

[34] C. Lee and M. Stoodley, “UTDSP benchmark suite,” 1998.

Anexo A

```
TranslationUnitDecl 0x7ffde98066d0 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x7ffde9806c08 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x7ffde9806920 '__int128'
|-TypedefDecl 0x7ffde9806c68 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x7ffde9806940 'unsigned __int128'
|-TypedefDecl 0x7ffde9806f98 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct
__NSConstantString_tag'
| `-RecordType 0x7ffde9806d50 'struct __NSConstantString_tag'
|   `-CXXRecord 0x7ffde9806cb8 '__NSConstantString_tag'
|-TypedefDecl 0x7ffde9807028 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x7ffde9806ff0 'char *'
|   `-BuiltinType 0x7ffde9806760 'char'
|-TypedefDecl 0x7ffde9807348 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct
__va_list_tag [1]'
| `-ConstantArrayType 0x7ffde98072f0 'struct __va_list_tag [1]' 1
|   `-RecordType 0x7ffde9807110 'struct __va_list_tag'
|     `-CXXRecord 0x7ffde9807078 '__va_list_tag'
|-UsingDirectiveDecl 0x7ffde9859e00 <taskgraph-tests/test2.cpp:4:1, col:17> col:17 Namespace 0x7ffde9807398
'std'
|-FunctionDecl 0x7ffde9859f20 <line:6:1, line:10:1> line:6:5 used function1 'int (int)'
| |-ParmVarDecl 0x7ffde9859e60 <col:15, col:19> col:19 used value 'int'
| `-CompoundStmt 0x7ffde985a1d0 <col:26, line:10:1>
|   |-DeclStmt 0x7ffde985a080 <line:7:2, col:7>
|   | `-VarDecl 0x7ffde985a020 <col:2, col:6> col:6 used x 'int'
|   |-BinaryOperator 0x7ffde985a148 <line:8:2, col:6> 'int' lvalue '='
|   | |-DeclRefExpr 0x7ffde985a098 <col:2> 'int' lvalue Var 0x7ffde985a020 'x' 'int'
|   | `-BinaryOperator 0x7ffde985a120 <col:4, col:6> 'int' '*'
|   |   |-IntegerLiteral 0x7ffde985a0c0 <col:4> 'int' 2
|   |   `-ImplicitCastExpr 0x7ffde985a108 <col:6> 'int' <LValueToRValue>
|   |     `-DeclRefExpr 0x7ffde985a0e0 <col:6> 'int' lvalue ParmVar 0x7ffde9859e60 'value' 'int'
|   `-ReturnStmt 0x7ffde985a1b8 <line:9:2, col:10>
|     `-UnaryOperator 0x7ffde985a198 <col:9, col:10> 'int' postfix '++'
|       `-DeclRefExpr 0x7ffde985a170 <col:9> 'int' lvalue Var 0x7ffde985a020 'x' 'int'
`-FunctionDecl 0x7ffde985a250 <line:12:1, line:38:1> line:12:5 main 'int (void)'
  `-CompoundStmt 0x7ffde902ea90 <col:12, line:38:1>
    |-DeclStmt 0x7ffde985a420 <line:14:2, col:12>
    | |-VarDecl 0x7ffde985a338 <col:2, col:9> col:9 used a 'double'
    | `-VarDecl 0x7ffde985a3a8 <col:2, col:11> col:11 used b 'double'
    |-DeclStmt 0x7ffde985a4a8 <line:15:2, col:12>
    | `-VarDecl 0x7ffde985a448 <col:2, col:6> col:6 used result 'int'
    |-ForStmt 0x7ffde985a6a0 <line:17:2, line:19:2>
    | |-DeclStmt 0x7ffde985a550 <line:17:7, col:17>
    | | `-VarDecl 0x7ffde985a4d0 <col:7, col:15> col:11 used i 'int' cinit
    | |   `-IntegerLiteral 0x7ffde985a530 <col:15> 'int' 0
    | |-<<<NULL>>>
    | |-BinaryOperator 0x7ffde985a5c8 <col:19, col:23> '_Bool' '<'
    | | |-ImplicitCastExpr 0x7ffde985a5b0 <col:19> 'int' <LValueToRValue>
    | | | `-DeclRefExpr 0x7ffde985a568 <col:19> 'int' lvalue Var 0x7ffde985a4d0 'i' 'int'
    | | `-IntegerLiteral 0x7ffde985a590 <col:23> 'int' 100
    | |-UnaryOperator 0x7ffde985a618 <col:28, col:29> 'int' postfix '++'
    | | `-DeclRefExpr 0x7ffde985a5f0 <col:28> 'int' lvalue Var 0x7ffde985a4d0 'i' 'int'
    | `-CompoundStmt 0x7ffde985a680 <col:33, line:19:2>
    |   `-UnaryOperator 0x7ffde985a660 <line:18:3, col:4> 'double' postfix '++'
    |     `-DeclRefExpr 0x7ffde985a638 <col:3> 'double' lvalue Var 0x7ffde985a338 'a' 'double'
    |-ForStmt 0x7ffde985a980 <line:21:2, line:23:2>
    | |-DeclStmt 0x7ffde985a768 <line:21:7, col:17>
    | | `-VarDecl 0x7ffde985a6e8 <col:7, col:15> col:11 used j 'int' cinit
    | |   `-IntegerLiteral 0x7ffde985a748 <col:15> 'int' 0
    | |-<<<NULL>>>
    | |-BinaryOperator 0x7ffde985a7e0 <col:19, col:23> '_Bool' '>'
    | | |-ImplicitCastExpr 0x7ffde985a7c8 <col:19> 'int' <LValueToRValue>
    | | | `-DeclRefExpr 0x7ffde985a780 <col:19> 'int' lvalue Var 0x7ffde985a6e8 'j' 'int'
    | | `-IntegerLiteral 0x7ffde985a7a8 <col:23> 'int' 20
    | |-UnaryOperator 0x7ffde985a830 <col:27, col:28> 'int' postfix '++'
    | | `-DeclRefExpr 0x7ffde985a808 <col:27> 'int' lvalue Var 0x7ffde985a6e8 'j' 'int'
    | `-CompoundStmt 0x7ffde985a960 <col:32, line:23:2>
    |   `-BinaryOperator 0x7ffde985a938 <line:22:3, col:11> 'double' lvalue '='
    |     |-DeclRefExpr 0x7ffde985a850 <col:3> 'double' lvalue Var 0x7ffde985a3a8 'b' 'double'
    |     `-BinaryOperator 0x7ffde985a910 <col:7, col:11> 'double' '*'
    |       |-ImplicitCastExpr 0x7ffde985a8c8 <col:7> 'double' <LValueToRValue>
    |       | `-DeclRefExpr 0x7ffde985a878 <col:7> 'double' lvalue Var 0x7ffde985a3a8 'b' 'double'
    |       `-ImplicitCastExpr 0x7ffde985a8f8 <col:11> 'double' <IntegralToFloating>
    |         `-ImplicitCastExpr 0x7ffde985a8e0 <col:11> 'int' <LValueToRValue>
    |           `-DeclRefExpr 0x7ffde985a8a0 <col:11> 'int' lvalue Var 0x7ffde985a6e8 'j' 'int'
    |-IfStmt 0x7ffde985ad20 <line:25:2, line:30:2>
```

```
|-<<<NULL>>>
|-BinaryOperator 0x7ffde985aa38 <line:25:6, col:10> '_Bool' '>'
| |-ImplicitCastExpr 0x7ffde985aa08 <col:6> 'double' <LValueToRValue>
| | `-DeclRefExpr 0x7ffde985a9b8 <col:6> 'double' lvalue Var 0x7ffde985a338 'a' 'double'
| `-ImplicitCastExpr 0x7ffde985aa20 <col:10> 'double' <LValueToRValue>
|   `-DeclRefExpr 0x7ffde985a9e0 <col:10> 'double' lvalue Var 0x7ffde985a3a8 'b' 'double'
|-CompoundStmt 0x7ffde985ac48 <col:13, line:28:2>
| |-BinaryOperator 0x7ffde985ab30 <line:26:3, col:9> 'double' lvalue '='
| | |-DeclRefExpr 0x7ffde985aa60 <col:3> 'double' lvalue Var 0x7ffde985a338 'a' 'double'
| | `-BinaryOperator 0x7ffde985ab08 <col:7, col:9> 'double' '+'
| |   |-ImplicitCastExpr 0x7ffde985aad8 <col:7> 'double' <LValueToRValue>
| |   | `-DeclRefExpr 0x7ffde985aa88 <col:7> 'double' lvalue Var 0x7ffde985a338 'a' 'double'
| |   `-ImplicitCastExpr 0x7ffde985aaf0 <col:9> 'double' <LValueToRValue>
| |     `-DeclRefExpr 0x7ffde985aab0 <col:9> 'double' lvalue Var 0x7ffde985a3a8 'b' 'double'
| `-BinaryOperator 0x7ffde985ac20 <line:27:3, col:8> 'double' lvalue '='
|   |-DeclRefExpr 0x7ffde985ab58 <col:3> 'double' lvalue Var 0x7ffde985a3a8 'b' 'double'
|   `-BinaryOperator 0x7ffde985abf8 <col:6, col:8> 'double' '/'
|     |-ImplicitCastExpr 0x7ffde985abc8 <col:6> 'double' <LValueToRValue>
|     | `-DeclRefExpr 0x7ffde985ab80 <col:6> 'double' lvalue Var 0x7ffde985a338 'a' 'double'
|     `-ImplicitCastExpr 0x7ffde985abe0 <col:8> 'double' <IntegralToFloating>
|       `-IntegerLiteral 0x7ffde985aba8 <col:8> 'int' 2
`-CompoundStmt 0x7ffde985ad00 <line:28:9, line:30:2>
  `-BinaryOperator 0x7ffde985acd8 <line:29:3, col:7> 'double' lvalue '='
    |-DeclRefExpr 0x7ffde985ac70 <col:3> 'double' lvalue Var 0x7ffde985a338 'a' 'double'
    `-ImplicitCastExpr 0x7ffde985acc0 <col:7> 'double' <LValueToRValue>
      `-DeclRefExpr 0x7ffde985ac98 <col:7> 'double' lvalue Var 0x7ffde985a3a8 'b' 'double'
-CallExpr 0x7ffde902e840 <line:33:2, col:13> 'int'
|-ImplicitCastExpr 0x7ffde902e828 <col:2> 'int (*)(int)' <FunctionToPointerDecay>
| `-DeclRefExpr 0x7ffde985add0 <col:2> 'int (int)' lvalue Function 0x7ffde9859f20 'function1' 'int
(int)'
`-ImplicitCastExpr 0x7ffde902e888 <col:12> 'int' <FloatingToIntegral>
  `-ImplicitCastExpr 0x7ffde902e870 <col:12> 'double' <LValueToRValue>
    `-DeclRefExpr 0x7ffde985ada8 <col:12> 'double' lvalue Var 0x7ffde985a338 'a' 'double'
-BinaryOperator 0x7ffde902ea10 <line:35:2, col:23> 'int' lvalue '='
|-DeclRefExpr 0x7ffde902e8a0 <col:2> 'int' lvalue Var 0x7ffde985a448 'result' 'int'
`-CStyleCastExpr 0x7ffde902e9e8 <col:11, col:23> 'int' <NoOp>
  `-ImplicitCastExpr 0x7ffde902e9d0 <col:17, col:23> 'int' <FloatingToIntegral>
    `-ParenExpr 0x7ffde902e9b0 <col:17, col:23> 'double'
      `-BinaryOperator 0x7ffde902e948 <col:18, col:22> 'double' '+'
        |-ImplicitCastExpr 0x7ffde902e918 <col:18> 'double' <LValueToRValue>
        | `-DeclRefExpr 0x7ffde902e8c8 <col:18> 'double' lvalue Var 0x7ffde985a338 'a' 'double'
        `-ImplicitCastExpr 0x7ffde902e930 <col:22> 'double' <LValueToRValue>
          `-DeclRefExpr 0x7ffde902e8f0 <col:22> 'double' lvalue Var 0x7ffde985a3a8 'b' 'double'
`-ReturnStmt 0x7ffde902ea78 <line:37:2, col:9>
  `-ImplicitCastExpr 0x7ffde902ea60 <col:9> 'int' <LValueToRValue>
    `-DeclRefExpr 0x7ffde902ea38 <col:9> 'int' lvalue Var 0x7ffde985a448 'result' 'int'
```